MASTER IN COMPUTER SCIENCE

Contribution to a database design workbench ADL - COBOL/GAM compiler

Cadelli, Mario; Muller, D.

Award date:
1985

Awarding institution:
University of Namur

Facultés Universitaires
Notre-Dame de la Paix
NAMUR

Institut d'Informatique

CONTRIBUTION TO A DATABASE DESIGN WORKBENCH

ADL - COBOL/GAM COMPILER

Mémoire présenté par
M. Cadelli / D. Muller
en vue de l'obtention
du titre de
Licencié et Maître
en Informatique
Année académique 1984-1985

Acknowledgements

First of all we would like to thank professor Hainaut, who conducted this thesis, for the interest he took in this work and for the help he gave us all along this year.

We are most grateful to professor Teichroew for welcoming us and for providing us with all the working facilities. We appreciate the opportunity he gave us to attend his classes at the University of Michigan.

We are also largely indebted to A. Delcourt and B. Van Houtte for their help in the development of the database framework.

Finally, many thanks to Michel Buyse who willingly put his tool at our disposal and supported us in the realization of the syntactical analyser. Last but not least, a special thank-you to Bruno Delcourt for his help and advice.

TABLE OF CONTENTS

INTRODUCTION

Generally speaking, the design of an information system (IS) is concerned with the specification of processes to be performed on data. Thus, an IS consists of processes specifications and data structures specifications.

As an IS may be very complex and its design rather complicated and time-consuming, automated tools are developed to help in the development of these systems. The integration of these automated tools form a whole software environment called SOFTWARE WORKBENCH. To be an efficient support for the development of IS applications, the software workbench needs to deal with both the processes and the data structures.

A software workbench is composed of a database of specifications as well as of general and specific tools. The general tools generally include some query and interactive updating facilities whereas the specific tools are for instance consistency validation tools and code generators.

The software workbench analyses the processes specifications. Its aim is to obtain programs and especially programs working on databases. They are the results of successive refinements. At the same time, the workbench allows one to get a database by successive refinements of the initial data structures specifications.

The subject of our thesis is the implementation of a specific tool of a workbench concerned with the design of database applications. It should perform the translation process of algorithms working on a database. So, in the first chapter of this document, we shall present the analysis and design methodology underlying the database design workbench. In order to set the overall framework for the work, we also introduce the components of the workbench. This introductory chapter ends with the description of the following chapters.

CHAPTER 1: A DESIGN METHODOLOGY FOR INFORMATION SYSTEMS

INTRODUCTION

From the EDP point of view, the prime characteristic of an Information System (IS) is the management and processing of common information shared between many users of a single system. Designed to act as a communication vehicle among its users, the IS mainly consists of two parts:

- a description of the organizational structure of the system it is representing;
- a specification of the processes which have to be executed on this data organization in order to make the IS behave like the real system of which it is a model.

Our introduction to the design methodology developed in Namur will mainly focus on the first component: the design of the IS database. Processes are only considered in what they concern the IS database. All this forms the subject of the first section of this chapter. More details about this general design methodology with examples can be found in [HAIN.83b], [HAIN.84b] and [HAIN.85].

A second section contains an overview of the logical database design workbench. The purpose of this section is to set a general framework and to show how our work becomes integrated into this workbench.

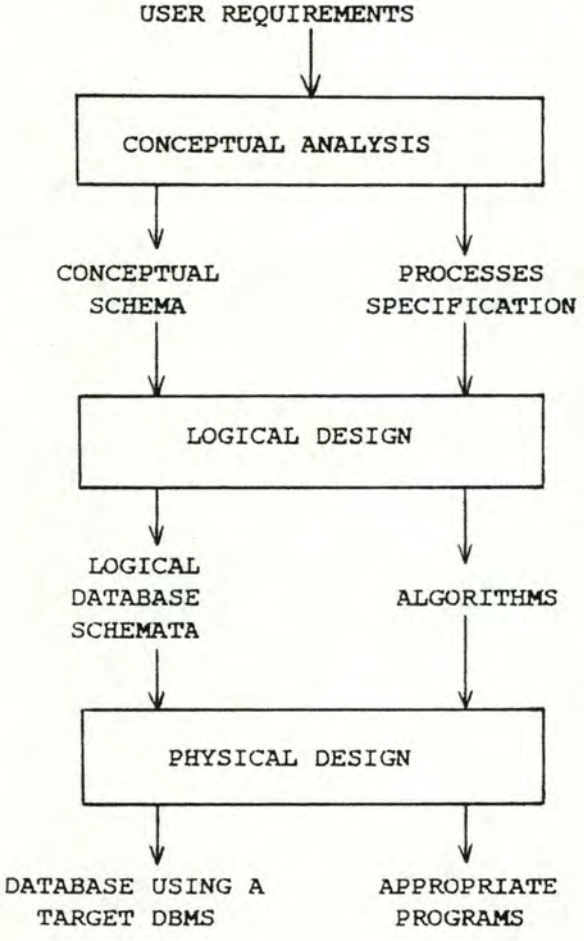
The last section describes the structure of this thesis by introducing the content of the following chapters.

1. THE DESIGN METHODOLOGY

The design of a database as part of the design of an information system is divided into three stages:

- the conceptual analysis;
- the logical design;
- the physical design.

Each phase consists of several steps characterized by specific by-products and it is supported by its own set of models. The figure grossly shows the relation which exists between these three design phases.



1.1. Conceptual Analysis

The conceptual analysis is devoted to the development of the conceptual schema of the IS. Derived from the user requirements and from the analysis of what exists, the conceptual schema constitutes an overall view of the IS. It includes the conceptual schema of the data, that of the processes and that of the dynamics. The details of this design phase are discussed in [BODA.83].

1.2. Logical Design

The second stage, the logical design, leads to the definition of an implementation of the data structures and of the requests on some abstract processor. The models used here are the GAM (Generalized Access Model) which describes the data structures and the ADL language (Access algorithm Description Language) which is used to express the algorithms. Thus, this implementation is done on a virtual processor which consists of a DBMS of the type GAM and of an ADL-language processor. The solution is constituted of:

- algorithms satisfying their specification and being efficient with regard to the number of accesses to the database;
- data structures representing the whole semantics included in the conceptual schema (and only this kind of semantics). These data structures include the access mechanisms judged necessary for the execution of the algorithms.

We can summarize the steps leading to these final products this way:

- the possible accesses schema (PAS)
  Using the GAM formalism, it expresses a data structure which is a direct representation of the conceptual schema. It contains any possible access to the data structure.
- the predicative algorithms
  For each request, the corresponding algorithm describes sets of data of the database by means of their properties; it does not describe explicitly a possible way of accessing the desired information. Each set of data is specified by a designation or selection condition called predicate.
- the effective algorithms
  Effective algorithms are functionally equivalent to predicative algorithms but they offer optimal performances in terms of the number of logical operations (logical record access). This performance criterion is necessary to choose among the various equivalent effective algorithms which can be derived from one predicative algorithm. The problem of the logical record access evaluation constitutes the subject of a former thesis ([DELC.84]).
- the necessary accesses schema (NAS)
  It expresses the data structures strictly necessary to the execution of the effective algorithms. It is a subset of the PAS. The NAS is obtained by retaining for each effective algorithm only the data structures and access mechanisms it uses. A subschema is thus obtained for each algorithm. The integration of all the subschemata gives the global effective schema.

1.3. Physical Design

In the final design phase, the logical solution defined in terms of an abstract ADL/GAM processor will be translated and optimized in order to adapt it to a real processor, that's to say to some existing programming language processor and Database or File Management System (DBMS or FMS).

Considering the constraints of the real data management system, the physical design phase derives from the NAS an access schema consistent with this system: the logical access schema. If some means of access has been eliminated in this process, the effective algorithms must be modified in consequence. The final steps (DBMS or FMS schema, application programs, external schemata, internal schema) constitute the concrete implementation of the IS (see [HAIN.83b]).

This mapping between the logical and the final physical solution (executable solution) can be achieved either by adapting the logical solution to the physical constraints of the real data management system or by implementing the abstract processor, in other words by making the logical solution executable (Access Module). The last option has the advantage of allowing the writing of algorithms which are independent of the real DBMS or FMS. They will only contain requests which correspond to some primitive functions of the virtual processor. However, it is only possible to implement the abstract processor if the translation of its actions into orders of the real data management system is systematic and can be automated.

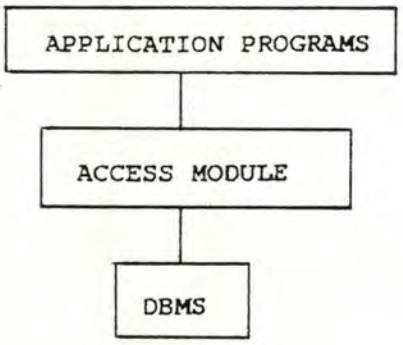
1.4. Access Module

We call ACCESS MODULE the module which is inserted between the application programs and a particular DBMS (or FMS), in order to hide the features specific of this DBMS. Through this architecture, the application programs are completely independent of the DBMS: a change of the DBMS version or even of the DBMS itself will only affect the access module.

In general, a program is independent of a certain type of modification, if this kind of modification in the data and their management system implies no modification of the source text of this program.

In the context of the design methodology, an access module which hides the physical implementation has to provide a virtual DBMS obeying the GAM specification. Among others, the idea of the database design workbench is to develop an access module which comprises also logical design steps: if possible it should allow the specification of predicative ADL algorithms which completely hide the choice of efficient access strategies.

The following figure shows the links between the access module and the application programs and the DBMS.



2. THE DATABASE DESIGN WORKBENCH

The term 'workbench' generally designates the development of a large software environment which should offer automated tools to increase the software production in a given domain. In our case, this domain is the design of database application programs and so we call the general framework: database design workbench. The purpose of this workbench is the development of a virtual processor, thus making it possible for database users to express their requests in terms of actions of this processor.

2.1. Representation Models

The implementation of this abstract processor is supported by:
- a database model;
- a language to express the requests.

A very important characteristic of these representation tools is their generality. Indeed, the usefulness of the implemented virtual processor depends upon the number of different real DBMS and language processors that can be represented by it. If its features only allow the translation to one particular DBMS, the access module offers little independence to the application programs: it merely hides the special syntax of the DBMS data description language and avoids the need of modification in the case of a version change.

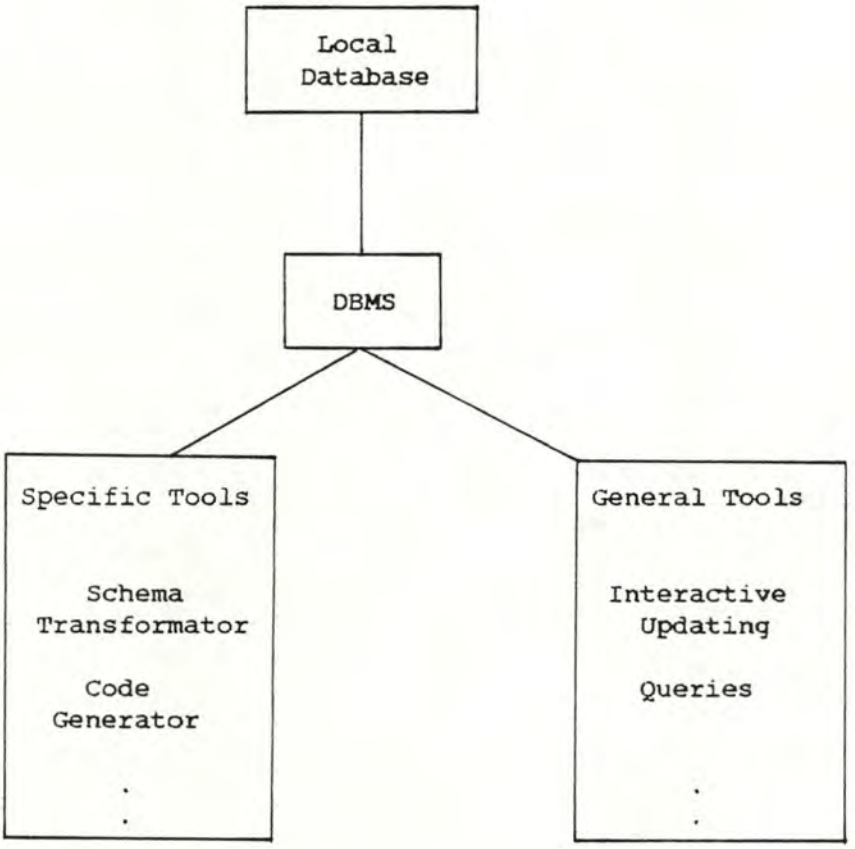
The objective of the database design workbench requires therefore that this data structure model provides all the concepts commonly used in the existing data management systems. Consequently, the workbench uses the Generalized Access Model (GAM), a model which offers a DBMS-independent expression of the data and emphasizes the access structures to the data. The objects, properties and primitive operations (access module) of this

model are described in the second chapter of this thesis.

As one of the purposes of the workbench is to achieve the automatic translation of predicative algorithms into efficient programs executable on a given DBMS, the algorithm description language must not only offer the power of usual programming languages; it needs to include the possibility of simply addressing collections of data in the database. The chosen language is called ADL, it offers the characteristics of a fourth generation language, that's to say procedural and declarative features.

2.2. Components of the Workbench

The workbench can be described by means of the following schema.



The workbench consists of a database containing all the informations necessary in the design process described above. It also consists of a DBMS and two kinds of tools. The general tools are, among others, an interactive query and an updating processor and a report generator. The specific tools are, for example, a schemata transformator, code generators and automated tools for the consistency validation of schemata and algorithms.

One of the aims of the database design workbench is the generation of an operational solution, that's to say a solution composed of a database schema and of programs, all executable on a real processor. This global design process has been decomposed into two major phases: the logical design and the physical design (see section 1).

For the first phase, the workbench needs some tools to achieve the schema and algorithm transformations described earlier in this chapter. The second phase is realized by means of the implemented virtual processor. It consists of a DBMS of type GAM and of an ADL-language processor. These various automated tools constitute the more specific components of the workbench.

The architecture adopted for the workbench is centered around the parse trees of the ADL algorithms. The source text of an ADL program is entered into the system (local database) via the ADL parser. Thus, it is represented within the system by its syntactical tree and all the consequent algorithm modifications necessary to obtain an effective ADL algorithm (see logical design phase) are performed by means of tree transformations. So, the parse tree represents the central part and the different specific tools are clustered round the tree structure. These tools are among others the semantic analyser to test the consistency of the database request with respect to the corresponding database schema, a compiler of effective ADL programs towards some conventional programming language, a decompiler to reconstruct an ADL text from the parse tree.

This general structure of the workbench indicates that the implementation of an ADL-Cobol/GAM compiler constitutes one of the specific automated tools. As this tool represents the first part at all of the ADL-processor to be implemented, its realization implies the definite definition of the language, the creation of an ADL parser, the development of the semantic analyser and finally the Cobol code generator. The modular architecture of the workbench determines this clear separation between syntactical analysis, semantical analysis and code generation process.

3. STRUCTURE OF THE THESIS

The introduction of the design methodology and the workbench allows us to define the rest of this work. The next chapter presents the concepts of the data model, namely the Generalized Access Model (GAM). We insist on its ability to represent a large range of data structures as it may be used to represent the semantics of a conceptual schema as well as to define the access mechanisms to the data.

In chapter three, we define the algorithm description language (ADL). We present in fact a subset of ADL, namely the syntax of effective ADL programs. But extensions towards a general ADL syntax are also specified.

Chapter four gives the conceptual schema of the database design workbench. The database of type GAM is necessary for storing the description of the database schemata of the application programs.

An overview of the software tools used for the implementation of the ADL parser is given in chapter five. These include among others a parser generator (YACC and LEX), tree construction functions and the global tool integrating all these tools used to generate an ADL parser.

In order to make sure that an ADL program is consistent with the description of the database schema it is working on, its database access requests have to be checked semantically. The principles of the semantic analyser and their implementation constitute the subject of chapter six.

Chapter seven discusses the principles of the Cobol text generation. It defines the choices which have been done to translate the syntactic structures of the ADL language.

Finally, the last chapter gives an example of a database schema and specifies the ADL text of one database request.

CHAPTER 2: THE GENERALIZED ACCESS MODEL

INTRODUCTION

The Generalized Access Model (GAM) is a descriptive formalism for data organizations. Thanks to its general concepts, the model offers the possibility to describe the "data organization" of any Data Base Management System (DBMS) or File Management System (FMS). The term "data organization" covers the data structure as well as the set of actions which can be performed on it, as the GAM has the quality of specifying the possible access structures to the data it describes.

Although the data management systems bear great differences, research has shown that they are based on a small number of common concepts. The GAM regroups all these widely used concepts. It presents their definition and properties regardless of any existing implementation.

Thus the GAM offers a general way to express the data and access structures necessary to a set of applications without doing reference to a particular DBMS or FMS. It is this property which recommands the use of the model in the logical design phase. The final matching between the logical data organization of the GAM and a particular DBMS or FMS can be provided by access interface modules.

This chapter contains:

- a description of the GAM objects
- an introduction to the data designation language
- a very short presentation of the main integrity constraints
- the definition of the actions offered to accede to and manipulate the data structures of the GAM.

A more detailed description of the subjects seen in the first three sections, including examples and a graphical representation of the GAM concepts, can be found in [DECL.84] and in [HAIN.84a]. All the same, we assume these conventions of representation to be known and we shall use them later on in the thesis. Lastly, we point out to you that a precise specification of the GAM basic operations is included in the appendix of the thesis.

1. THE GAM OBJECTS

The objects of the model are the records and record types, the data item values and data items, the access paths and access path types, the files, the database, the access keys and orders and the record references.

1.1. Record and Record Type

A RECORD is a set of information which can be created, acceded to, updated and deleted. It is the logical means of communication between a program and the database.

A record belongs to one and only one type. The RECORD TYPE defines the common properties of a class of records. It has a name which identifies it within the database. At a given time, 0, 1 or more records belong to a record type.

1.2. Data Item Value and Data Item

A DATA ITEM VALUE is an element of a given data type (for example string, integer number) which can be associated to a record. Being a data type value, the data item value is a piece of information represented by a string of symbols. The access to the data item values is possible by acceding to the record they are attached to. Data item values cannot be created or deleted; they can only be attached to a record or detached from it. At a given time, 0, 1 or more data item values can be associated to a given record.

A DATA ITEM denotes the association of a data type to a record type. It has a name which identifies it among the data items of the record type. Indeed a record type may contain 0, 1 or several data items of various types.

Among the data items, we distinguish between:

- ELEMENTARY and DECOMPOSABLE data items.
  An elementary data item defines a set of atomic values. This means that it is impossible to further decompose these values into meaningful fragments of information. A decomposable data item however denotes a set of values which contain several meaningful data segments. The component data types of a decomposable data item are considered to be data items themselves, as they are recursively elementary or decomposable.

- SIMPLE and REPETITIVE data items.
  A data item is simple if one and only one value of this data item is associated to each record. A data item is repetitive if more

than one corresponding data item value may be attached to a record.

1.3. Access Path and Access Path Type

An ACCESS PATH is a mechanism which links a record (called ORIGIN) to zero, one or more records (called TARGETS) so that the targets are accessible from their origin. It defines a one-directional access.

Each access path belongs to one and only one type. The ACCESS PATH TYPE defines the common properties of the class of access paths it represents. An access path type is characterized by its origin and target record types. A name is associated to an access path type to identify it among all the path types which have same origin and target. This does not exclude the possibility of an 'empty' name.

An access path type AP2 is the INVERSE of an access path type AP1 if, for each target T linked to the origin O by the access path type AP1, there exists an access path type AP2 linking T to O.

1.4. File

A FILE is a dynamic collection of records. Each record belongs to one single file, but a given record type may be collected by several files. Finally, a file may collect more than one record type.

1.5. Database

We give here the definition proposed by Moentack & Delcourt ([DELC.84]):

> " A DATABASE is a shared collection of interrelated data designed to meet the needs of multiple types of users. Logically, it is the record collection of a set of files and the associated data structures."

1.6. Access Key

Composed of one or several data items of a record type, the ACCESS KEY represents a mechanism to reach the elements of a selected record set sequentially. The criterion of selection is to consider only those records which are associated to a given value of this access key.

An access key is characterized by a reference set. It determines the set of records to consider when applying the selection criterion of the access key. This reference set may be all the records of the database, all the records belonging to one file or all the records target of a particular access path type.

1.7. Order of a Sequence of Records

The concept of "sequence of records" has been touched on several times. Databases, files, access path targets as well as sets of records selected by an access key constitute sequences of records. The access to these sequences is done according to some implicit or explicit order. This order is said to be simple or global.

Both type of orders may define:

- a random order (no order),
- a chronological (FIFO) or non-chronological (LIFO) order for the insertion into the sequence
- a programmed ordèr, leaving the determination of the place of insertion to the application programs (insert PRIOR or NEXT to some record)
- a sorted order using the values of some data item or group of data items.

1.8. Record References

The concept of RECORD REFERENCE offers the possibility of addressing database records at a logical design level. The record reference is introduced into the GAM in order to make the manipulation of record addresses completely independent of the data management system which is used to implement the logical data organization described in terms of the GAM. We shall come back to this notion later in this chapter.

2. THE DATA DESIGNATION LANGUAGE

The Data Designation Language (DDL) allows to express selected sets of records and data item values by defining their properties. It is introduced to simplify the formal expression of some integrity constraints and to set the context for the Access algorithm Description Language (ADL), as it constitutes the basis of this language (we describe the ADL in chapter 3). The presentation we shall give of the DDL is quite incomplete. More details can be found in [HAIN.84a], [DELC.84] and somewhat [HAIN.83a]. These references were the basis for the realization of this part. Finally, note that the examples given for the purpose of illustration refer to the database schema presented in the example of chapter 8.

2.1. Notations

Record types, access path types and data items are represented by their names.

2.2. Sets

We consider sets of records and sets of data item values. Both set types can be described by simply specifying the name of the type (record type or data item) they belong to, or by additionally giving a general selection condition. For sets of records it is also possible not to specify the record type name but the name of a record variable which references the selected records. Furthermore a most general set of records, including records of different types, may be denoted by the string "RECORD".

General syntax:

‹name› [/‹var›] ‹condition›

N.B. The bracket symbols are used to denote an option.

- ‹name› is a record type, a data item, a record variable or the reserved name RECORD.
- ‹var› is the name of a record variable; this option can only be used in the expression of a set of records.
- ‹condition› is the expression of a condition (see section 2.3.); it may be empty.

This expression semantically represents the instances of a record type or data item which satisfy the condition. If the option "/‹var›" is used, this variable may from now on be used to name the records of the selected set.

Examples:

CUSTOMER : represents all the records of type CUSTOMER stored in the database.

ORDER /ORD(NB-ORD=123) : represents the records of type ORDER which have an order number equal to 123; the variable ORD will denote this set of records.

NAME(:CUS) : if the record variable CUS denotes a record of type CUSTOMER, this expression gives the value of the data item NAME associated to this record.

2.3. Conditions

A condition is a simple condition or a boolean expression of simple conditions. By associating a condition to the expression of a set, one defines a subset of the set. The set of elements for which the condition is evaluated is called 'qualified set' and by "elements" we mean records or data item values. In this section we shall confine ourselves to presenting two types of simple conditions: the relation conditions and the belonging conditions.

2.3.1. Relation conditions

An element of the qualified set will be selected according to the number and the properties of the elements related to it.

General syntax:

(‹relation›:‹cardinal›‹related-set›)

- ‹relation-set› is the expression of a set of elements
- ‹relation› is the name of a relation defined between the types TNAME1 and TNAME2 where TNAME1 is the type name of the qualified elements and TNAME2 is the type name of the elements belonging to ‹related-set›. The relationship existing between a record type and a data item has an empty name.
- ‹cardinal› is the expression of an integer interval:
  [1..3] represents the integers 1,2,3.
  [0..*] represents the whole set of positive integers
  [1..1] represents the integer 1
  an empty cardinal stands for [1..*].

Semantic interpretation:

An element of the qualified set is selected if it is related by ‹relation› to I elements of the ‹related-set› with I belonging to ‹cardinal›.

Examples:

CUSTOMER(C-OR:ORDER) represents the records CUSTOMER associated by C-OR to at least one ORDER.

NAME(:CUSTOMER(C-OR:ORDER)) represents the values of the item NAME of the above CUSTOMER records.

ORDER(OR-OL:[3..6]ORDER-LINE) represents the records ORDER which are related by OR-OL to more than 2 and less than 7 records ORDER-LINE.

2.3.2. Belonging conditions

An element of the qualified set will be selected according to its belonging or non belonging to a collection of elements of compatible type. The belonging condition mainly serves to apply a selection criterion on data item values.

General syntax:

<rel-op><S> and (<rel-op><S>)

- <rel-op> is a relational operator (=, <, <=, >, >=, <>, in, not-in)
- <S> is the expression of a set of elements. The type of these elements has to be compatible with the type of the qualified elements.

Semantic interpretation:

An element of the qualified set is selected if <S> and the set formed by the element satisfy the relation specified by <rel-op>.

Examples:

Q-STK>10 : the values of the item Q-STK that are higher than 10 satisfy the relation specified by <rel-op>.

PRODUCT(:Q-STK<10) : the expression selects the records PRODUCT with a stock quantity lower than 10.

Remark:

As the conditions are expressed on sets of elements, their expression may require the use of some functions which, for instance, give the size of a set, the rank of an element in the sequence. Some useful functions are described on page 2.11 of [DELC.84]. Moreover, M. Hainaut in [HAIN.84a] presents some extensions to the condition expressions explained in this section.

3. INTEGRITY CONSTRAINTS

The purpose of the GAM is to offer a DBMS-independent expression of some data organization. But the basic concepts of the model introduced in section 1 give a too general representation of the data structures derived from some conceptual specifications. Indeed these structures must satisfy certain rules called integrity constraints.

To allow an equivalent logical representation with the GAM concepts, we add to the model expression a set of rules corresponding to the conceptual integrity constraints. They are expressed with the Data Designation Language (see section 2) and some widely used constraints like functional class, existence constraint and identifier, are integrated as additional concepts into the GAM. This section contains a short description of these complementary GAM concepts. If you want formal definitions and examples, we advice both [DELC.84] and [HAIN.84a]. The last reference has the advantage of presenting still other useful constraints.

3.1. Functional Class

The functional class property applies to relation types. Suppose R is a relation type defined between the object types (record types or data items) A and B. The functional class property determines the maximum number of objects of B (or A) that may be associated to one A (or B). Functional class is an oriented concept because of the non symmetric mathematical concept of "function". Therefore in the following, the relation type R is assumed to have as origin A and as target B.

One-to-many relation type : 1-N
The functional class of R is one-to-many if for each object B, the maximum number of objects A associated to it by R is one.

Many-to-one relation type : N-1
The functional class of R is many-to-one if for each object A, the maximum number of objects B associated to it by R is one.

Many-to-many relation type : N-M
The functional class of R is many-to-many if any number of A objects (respectively B objects), may be associated by R to one object B (respectively one object A). There is no constraint.

One-to-one relation type : 1-1
The functional class of R is one-to-one if the maximum number of objects of type B associated to one object A is one and if the maximum number of objects of type A associated to one object B is one.

Of course the one-to-one relation-type is the most restrictive. It is both a many-to-one and a one-to-many relation type. The last two classes of relation types are again a subset of the many-to-many relation type.

3.2. Identifier

Suppose a relation type R having as origin the object type A and as target the object type B. An object B identifies the object A associated to it by R if there is no other object of type A related to this same object of B. In other words a target object of a one-to-many or one-to-one relation type identifies the corresponding origin object.

An identifier of an object type A is simple or multiple depending on wether it is composed of one object type B (see above) or of a group of object types B1, B2, .. , each of them associated to the object type A by a different relation type (R1, R2, ..).

3.3. Existence constraint

This constraint only applies to record types, member of some relation type. If a record type is subject to the existence constraint, each instance of it has to be associated by the specified relation type to an instance of the related object type (record type or data item). In other words this constraint requires the association between the constrained record type and the corresponding object type at any time.

4. THE GAM FUNCTIONS

To be complete the description of a data structure should include the set of operations allowed to be performed on it. So by 'GAM functions', we mean the basic operations defined on the data and provided to the user of the database. They constitute the access or interface module.

We separate the GAM functions into two classes: those which accede to the data and those which update or modify them. Each of these classes forms the subject of a section. A third section is dedicated to the particular problem of the record reference handling; it introduces the few functions added to the access module in order to standardize the format of the record variables (see section 2) within the application programs.

Furthermore we want to remind you that a detailed definition of these routines is given in the appendix. For this part, a useful reference has been once more [HAIN.84a] and also [HAIN.81]. The actual design of the interface functions however is largely inspired by the work of Bock and Delval, two former students [BOCK.82].

4.1. The Data Access Functions

4.1.1. Principles

The purpose of an access is to make available to a user one or more records stored in the database. So, the access often represents only the first of a database action like the extraction or the modification of some data item values.

Logically we may define any type of access to records of the database as being an access to some sequence of records. Each access function can be defined by :

- the sequence of records on which it operates
- the position within the sequence of the wanted record.

Sequence of records

When describing the DDL and in particular the expression of a set of records (see section 2.2.), we have seen that a sequence of records is determined by :

1. the type of the records :
   - the records of a single type
   - all the records independently of their type

2. the selection condition (see section 2.3.) :
   - all the records without any restriction (empty condition),
   - the records satisfying some condition to which corresponds an access mechanism. This mechanism leads to the selection :
     - access by key
     - access within an access path
     - access to the records of a file
     - combination of these
   - the records satisfying some condition not necessarily corresponding to an access mechanism, like a condition on the data item values of a record or a condition of belonging to an access path. However these complex record selections cannot be realized by the basic operations of the access module.

A sequence of records always has an order (see section 1.7.). The order in getting the records may be :

- random,
- natural (that's to say the order of the subset from which the records are extracted),
- predefined (generally sorted).

Position of a record

The position of a record can be specified by its rank or by its relative position within the sequence :

- first record of the sequence
- record following a particular record of the sequence
- last record of the sequence
- record preceding a particular record of the sequence
- record of a given rank (i-th) up from the first or from the last record of the sequence.

4.1.2. Functions

4.1.2.1. Access to the database

The two operations delimit the time the user works on the database.

1. Opening of the database

The purpose of this function is to make the database accessible to the user. Thus it opens all the files of the database or if the database is already open, it reports this fact to the user. In this last case, the user should avoid closing the database at the end of the application.

As we consider the database the user is working on to be a local database, there is only one. Therefore the function needs no input parameter and it returns the result of the request.

2. Closing of the database

This function makes the database unavailable to the user by closing all its files. It returns an error code to the user if the database was not open. This function also has no input parameter.

4.1.2.2. Sequential access to the records of one type

This function allows one to sequentially get all the instances of some record type of the database. As input this function requires :

- the type of the record,
- the reference of the record previously acceded,
- the order if any predefined order is wanted.

The access operation consists in providing the reference of the record which immediately follows the instance specified by the input with respect to the defined order if any has been specified. If the reference should be a 'null' value, this function gives the first record in the sequence according to the defined order.

If the access fails however, the function reports the reason for it, like the reaching of the end of the records sequence or the incorrectness of one of the input parameters.

4.1.2.3. Access by key to the records of one type

This function allows one to reach the records associated to a specified value of some access key, one at a time. As this selection criterion defines a sequence of zero, one or more records of a given type, this function needs as input information :

- the record type,
- the reference of the record previously acceded if any,
- the access key which is used,
- the type of the test operator (presently only the exact match ("=") is allowed),
- the value of the access key.

The successful execution of this routine provides the reference of the record which satisfies the access key condition and immediately follows the record defined by the two first input informations. If the reference contains a 'null' value, the function provides the first record of the selected sequence. The order of the records in the sequence is defined at the declaration of the access key as an IKO (see chapter 4) and it determines the order of extraction.

If the access fails, the function reports to the user or application program the reason of the failure, like the incorrectness of an input or the non-existence of the desired record.

4.1.2.4. Access to the target records of an access path

This function allows the sequential access to the records linked by some access path to a specified origin record. Given the four kinds of access path types, the sequence of the target records may be composed of zero, one or more records. Thus the request of a sequential access within some access path must specify :

- the type of the access path,
- the reference of the current origin record of this access path type,
- the reference of the previously acceded target, if any,
- the order if any predefined order is wanted.

The output will be the reference of the target record which is the first to follow the previously provided target in the selected sequence according to some natural order or to the order specified in the access request. The function also allows the access to the first or to the only record (for a one-to-one or many-to-one access path) of the sequence through the possibility of specifying a 'null' value for the reference of the previous target. As the GAM allows access path types with multiple target record types, the output includes the specification of the type of the reached record.

Moreover the function states the way the operation has been concluded. In the case of a failure, because the end of the sequence of targets is reached or because an inconsistency among the inputs with regard to the database schema has been detected, the access function reports the exact cause of the failure.

4.1.2.5. Access by key to the targets of an access path

This function is a combination of the two previous ones; it allows the access on a key for an access key defined within an access path. As this access key is not necessarily an identifier for the target record type and as the access path logically has to be one-to-many or many-to-many, the set of selected records is again a sequence of zero, one or more records. Therefore the access routine needs as input:

- the type of the record you want to accede,
- the access key which is used,
- the type of the access path the access key is defined in,
- the reference of the current origin record of this access path type,
- the reference of the previously acceded target, if any,
- the type of the test operator (presently only the exact match is allowed),
- the value of the access key.

If the access succeeds, the function provides the reference of an instance of the specified record type. This record is associated to the value defined for the access key. It is linked to the indicated origin

record by the given access path type and it directly follows the previously acceded record. A 'null' reference will get the first or the only record of the selected sequence of targets. The order of sequence is either defined by a natural default order or by the access key, if the key is at the same time an order (see IKO in chapter 4).

In case of a failure, the access routine will indicate the exact cause: if there is no record matching the access condition or if one of the inputs is incorrect.

4.1.2.6. Access by reference to a record of the database

The purpose of this function is to provide the data item values and/or the type of the record corresponding to the indicated reference. The information to be provided is therefore the reference and the record type if it is known.

4.1.2.7. Access to the data item values of a record

Logically a record exists independently of its data item values as they are associated to one another by an access path without a name. It would therefore be possible to handle this access as a normal access within a path, but this option seemed to us inefficient. Indeed acceding each data item value separatly would necessitate a large number of access requests. We decided to integrate the access to the data item values in the types of access described above. Thus every function that provides the reference of a record includes an indicator specifying if all or no data item values of this record should be made available. Presently this indicator is even assumed to always ask the transfer of the data item values.

4.2. The Modification Functions

4.2.1. Principles

The purpose of a modification function is to update some data so that the complete data structure continues to be a correct picture of the system it represents. So considering the basic elements of the GAM, the necessary operations are :

- the creation of a record,
- the deletion of a record,
- the modification of the data item values asociated to a record,
- the insertion of a record as target into an access path,
- the retrieval of a target record from an access path,
- the transfer of a record from one access path to another of the same type.

However the data is generally subject to some integrity constraints (see section 3) which still have to be respected in the new state of the database. Consequently the primitive actions stated above will have to include some integrity checking. In case of a request leading to the violation of some constraint, they could theoretically either refuse the operation or execute it, but with a series of other updatings in order to reobtain a coherent state. Both options are valid, but for the second case the supplementary actions must be precisely defined. All the same, the execution of such a basic operation has to be an atomic action for the user: during its execution the database is blocked for every user at it passes through incoherent states.

The number of possible integrity constraints being quite large, the basic modification operations will only take into account the most frequent ones: the functional class property of an access path, the identifier condition of some data items or record types and the existence constraint of a record type (see section 3). By this we are well aware of not handling the problem of redundant data. The responsability of maintaining an equivalence between the redundant data structures, which are introduced for reasons of performance or security, is left with the data management system the GAM is based on.

4.2.2. Functions

4.2.2.1. Creation of a record

The purpose of this function is to create a new instance of some record type provided that the informations communicated by the request are sufficient to verify that the integrity of the database will not be affected. The input of the function must specify :

- the type of the record to be added to the database,
- the list of its data item values,
- the list of the access path types this record should be attached to as a target,
- the reference of the origin records of these access path types.

The integrity of the database first requires that the data item values of the record to be created are valid (do not violate an identifier constraint). Then the indicated list of access path types must only denote access path types which indeed contain the record type as a target. But at the same time, the list must contain all the access path types for which the record type is a mandatory target member (existence constraint). Finally the corresponding record references, which are given in order to identify some particular instance of the access path types, have to reference origin records which may still be associated to another target (functional class constraint).

If all these conditions are satisfied, the creation function stores the new record with its data item values into the database and links it to

all the origin records via the specified access path types. To the user or application program, it returns the reference of the created record and reports the success of the operation. If one of the integrity constraints is violated, the creation is refused and the reason of the failure is reported to the user.

4.2.2.2. Deletion of a record

The effect of this function is to remove a given record from the database making it definitively inaccessible. This operation may become very complex if the record to be deleted is still origin of some access path types, provided that one takes the option of executing the request all the same.

As input information the deletion operation needs :

- the type of the record,
- the reference of the record.

The function begins by detaching the record to be deleted from all the access paths in which it is a target. It then detaches all the optional targets from the access paths of which the record to be deleted is origin and it recursively applies this complete process of deletion to the mandatory targets. At last it deletes the record from the database. So, asking the deletion of one record may cause the deletion of a large number of associated records. The user should therefore be well aware of the effects of this kind of request.

The function terminates by reporting the successful execution of the deletion or, in case of an incorrect record type or record reference, by giving the exact reason of the unsuccessful deletion.

4.2.1.3. Modification of the data item values of a record

The purpose of this operation is to update some data item values associated to a given record. To execute the updating, one needs to identify the record as well as to know the data items the new values will be assigned to. Therefore the modification function asks for :

- the type of the record to be updated,
- the reference of this record,
- the list of the data item values.

It doesn't need the list of the particular data items which will be updated, as it is assumed that the list contains all the data item values of the record and not only the new values. This choice was done to allow a fixed structure of the values communication zone; it always has the structure of the record which is currently implied in the operation.

However the special meaning of some data items associated to a record type makes it necessary to distinguish two types of updating :

- the modification of the values of simple data items of the record
- the modification of the values of data items representing some identifier, access key or order, in other words, data items which are components of some IKOs.

This clear separation is done as the updating of the value of data items of the second type involves more integrity verifications (the non-violation of an identifier condition) or some additional actions (storage in a sorted order).

Both functions will only consider the data item values they are allowed to change. They store the record with these updated values into the database, on condition that no integrity constraint will be affected. Finally they report the successful execution of the operation or the reason of the failure.

4.2.2.4. Insertion, retrieval or transfer

This function combines three types of operations :

- the insertion of a defined record into a particular access path,
- the retrieval of a target record from a particular access path,
- the transfer of a target record from one access path to another of the same type.

The last operation represents the combination of the two others, with the only difference that for the user it constitutes one atomic action. As a matter of fact, the existence constraint of some record types makes this transfer operation necessary, as a retrieval request with a subsequent insertion would be refused: the integrity of the database would be violated between the two operations.

But apart from this distinction, the three operations seemed to us so similar that we decided to regroup them into one single function. It is the presence or absence of some input parameters (reference of the old origin, reference of the future origin) that decides which one of the three operations will be executed.

Note that in the case of an access path of the functional class one-to-many, the reference of the possible former origin record must not be indicated as it is implicit. If the record implied in the operation is asked to be attached to some new origin record, the integrity constraint imposes that, at the same time, it is detached from the record it was previously associated to by an access path of this type. So, it is the belonging or not belonging of the record to an access path of the considered type that decides if the operation will be a simple attach or a transfer request.

Obviously, if no reference of an origin-to-be is indicated, the operation is assumed to be a single detach request.

It follows that this multiple function needs as input :

- the reference of the record to be moved from one access path or/and to be attached as target to an access path,
- the type of the access path(s),
- the reference of the origin record from which the record should be detached if the access path is many-to-many,
- the reference of the origin record of the access path to which the record should be attached as a target.

According to the presence of the last two parameters and to the functional class of the access path type, the function determines the desired operation. The operation is executed on condition that no existence constraint will be violated (detach in a mandatory access path). At last the function reports either the success of the execution or the cause of the failed execution to the user or application program.

4.3. The Record References Handling Functions

4.3.1. Principles

The purpose of these functions is to provide a standardized method for handling record references in an ADL program. This problem of the reference to a database record arises as the several existing DBMS offer quite different ways for directly addressing a stored record. For instance, in a DBMS Codasyl, a record reference is represented by the database key. In the former Codasyl versions, it was possible to use the database key directly, provided that a variable of the adequate type format had been declared. The last version however, introduces the concept of a 'keeplist', thus making it impossible to directly manipulate record addresses in the application programs. As a matter of fact, the keeplist represents a level of indirection as the address communicated to the user does not represent the actual address of a record, but an index value into this keeplist where the actual address can be found.

The main purpose of the logical design tools (GAM and DDL) is to allow a DBMS-independent expression of the data structures. Considering only the case of the Codasyl systems, it appears that this independence requires the inclusion of reference handling functions into the access module. These functions manage some sort of keeplist. So each initialization or annulment of a reference variable, each comparison between two reference variables as well as each assignation of a record reference of one reference variable to another must be done by means of these functions.

4.3.2. Functions

4.3.2.1. Comparison of two reference variables

This function compares two record variables and states the result of this comparison. It tells the user or application program if both variables reference the same record or not.

4.3.2.2. Annulment of a reference

The purpose of this function is to change the value of the reference variable specified in the input in such a way, that it no longer references any record in the database.

4.3.2.3. Assignation of a record reference from one variable to another

This function is used to assign the record reference of some record variable to some other record variable. So, given both reference variables (the target and origin reference variable), the operation consists in modifying the target reference variable in such a way, that from now on it references the same database record as the origin reference variable.

CHAPTER 3: THE A D L DEFINITION

INTRODUCTION

Like the Generalized Access Model (GAM, see chapter 2) is necessary to allow a DBMS-independent expression of the data structures, the Access algorithm Description Language (ADL) is necessary in a process design methodology to allow a programming-language-independent expression of the algorithms. The language has been designed to facilitate the access to complex data structures as well as their management. Based on the Data Designation Language (see section 2 of chapter 2), it allows the description of all types of access, from the most complicated ones (selective access) to the punctual ones which are provided by the access module (see section 4 of chapter 2).

This generality of the language made it necessary to distinguish two levels in the ADL :

- a subset of the ADL which only allows the expression of effective algorithms;
- the complete ADL which allows the expression of all kinds of algorithms, effective and predicative.

Whereas an effective algorithm may only contain access requests which correspond to some basic access function of the access module (see section 4.1. of the previous chapter), a predicative algorithm is allowed to define all sorts of complex access expressions. It merely states the selection criteria of the records to be manipulated.

The purpose of the database design workbench is to allow the compilation of predicative ADL algorithms. So, on one hand, the syntactic analyser has to parse programs written in the general ADL and, on the other hand, the ADL compiler has only to cope with effective ADL algorithms. Consequently, this chapter contains the defining description of the subset-ADL (or "effective" ADL) as well as extensions which have already been realized for the syntactic analysis.

In more details, the first part of the chapter will begin by introducing the general features of the language and by explaining the notation formalism of the syntax. Then, in a first section, we shall define the basic tokens of the language to continue, in the following section, with describing the variables allowed in the subset-ADL. The third section presents the rules and the meaning of the ADL expressions. These are the collection expression, the arithmetic expression and the general expression. In the fourth section, the structure of an ADL program is defined. The next section will then explain in detail the way of declaring data types and variables in an ADL program. At last, the sixth section lists the operational units of the language, known as statements.

In the second part of the chapter, we shall describe in detail some extensions to this effective ADL. We are well aware that these syntactic extensions are yet insufficient to obtain a completely general ADL definition. The lack of generality concerns in particular the condition

expressions.

Finally, we like to mention that our ADL definition is largely based on a previous but incomplete definition of the language. It was done by two former students, Moentack and Delcourt, and it can be found in their thesis [DELC.84]. It follows that some parts are exactly alike, but we decided to explain them again in order to present a complete definition in our work. Moreover, a complete description of the language in BNF (Backus-Naur Form) is provided in the appendix.

1. THE EFFECTIVE ADL DEFINITION

Designed to allow a simple expression of database manipulation algorithms, the ADL contains not only conventional programming language features of computation and control structure, but also particular database access components. The calculating rules are expressed by the concept of arithmetic expression. As means for defining an algorithmic structure, the ADL includes the alternative statement, the iterative loop and the function call. To represent the database access expressions we use the concept of DB_object_set.

From a global point of view, the ADL syntax distinguishes two main parts in an algorithm: the specification of the actions to be executed and the declarations. The declaration part supports the operational part (called statement part) as it gives information about the properties of the objects that will be used in the statement part. By defining the type of an object, the declaration determines where and how it may be manipulated. In particular, the declaration delimits the class of values which can be assigned to the object. By the constraint that all objects used in the program have to be defined, the compiler is able to test the existence of the manipulated quantities.

Notation Formalism for the Syntax

The syntax is described with the help of metalinguistic formulas. We best describe their interpretation by giving an example.

‹ab› ::= ‹d› | (‹ab›) | ‹ab›‹d›

A sequence of characters enclosed between the symbols ´‹ ›´ represents a metalinguistic variable whose values are sequences of symbols. The marks ´::=´ (definitional sign) and ´|´ (alternative sign: logical OR) are metalinguistic connectives. Any symbol in a formula which is neither a variable nor a connective denotes itself. Juxtaposition of symbols and/or variables in a formula means juxtaposition of the sequences denoted. Hence the formula above gives a recursive rule for forming the values of the variable ‹ab›. It indicates that ‹ab› may be formed by some value of the variable ‹d›, by some legitimate value of ‹ab› enclosed between brackets or by some value of ‹ab› followed by some value of ‹d›. If the values of ‹d› are decimal digits, some possible values of ‹ab› are :

1234
(12345)
((9)8)

In order to improve the readability of the syntax definition, we have chosen to distinguish the metalinguistic variables by using words which convey approximately the nature of the corresponding variable. If these same words appear somewhere else in the text, they will refer to the corresponding syntactic structure.

1.1. Basic Symbols, Names, Numbers and Strings

1.1.1. Basic Symbols

The ADL language is built upon the following basic symbols :

```
<basic_symbol ::= <letter>
                | <digit>
                | <logical_value>
                | <delimiter>
```

1.1.1.1. Letters

```
<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
```

Letters do not have any individual meaning. They are used for forming names and strings.

1.1.1.2. Digits

```
<digit> ::= 0|1|2|3|4|5|6|7|8|9
```

Digits are used for forming numbers, identifiers and strings.

1.1.1.3. Logical values

```
<logical_value> ::= true|false
```

The logical values have a fixed obvious meaning.

1.1.1.4. Delimiters

```
<delimiter> ::= <operator>
              | <separator>
              | <bracket>
              | <declarator>

<operator> ::= <arithmetic_operator>
             | <test_operator>
             | <logical_operator>
             | <sequential_operator>
             | <relationship_operator>

<arithmetic_operator> ::= +|-|*|/|^
```

```
<test_operator> ::= <|>|<=|>=|<>|=

<logical_operator> ::= and|or|not

<sequential_operator> ::= for|while|if|next|exit
                          |create|modify|delete|return

<relationship_operator> ::= (:

<separator> ::= ,|.|;|:=|..|do|then|else|endfor|endif|endwhile

<bracket> ::= (|)|[|]|begin|end

<declarator> ::= algorithm|type|var|group|array|of|items_of
                 |ref|boolean|string|real|integer|numeric
```

Delimiters have a fixed meaning which is obvious in most cases. If not, the meaning is explained at the appropriate place in the sequel.

Blank space and carriage return characters may be freely used to improve the presentation and the readability of an ADL program, except between keywords and names. Moreover, it is possible to include some text explanations between the symbols of a program : a text ADL may contain comments which respect the following convention :

```
/* <any sequence of characters not containing '*/'> */
```

1.1.2. Names

1.1.2.1. Syntax

```
<name> ::= <name> <letter>
         |<name> <digit>
         |<name>_
         |<letter>
```

1.1.2.2. Examples

```
I
CUSTOMER
NUM_PROD
```

1.1.2.3. Semantics

Names have no inherent meaning. Their purpose is to help identifying data types, simple or group variables, arrays, record variables, functions and the algorithm itself. The same name cannot be used to denote two different quantities.

1.1.3. Numbers

1.1.3.1. Syntax

```
<unsigned_integer> ::= <unsigned_integer> <digit>
                      |<digit>

<decimal_number> ::= <unsigned_integer>.<unsigned_integer>

<sign> ::= +|-

<exponent_part> ::= e <sign> <unsigned_integer>
                   |e <unsigned_integer>

<unsigned_real> ::= <decimal_number> <exponent_part>
                   |<decimal_number>
```

1.1.3.2. Examples

```
31
3.1
0.31e-3
```

1.1.3.3. Semantics

Decimal numbers have their conventional meaning. The exponent part expresses an integral positive or negative power of ten.

1.1.4. Strings

1.1.4.1. Syntax

<string> ::= ´ <any sequence of basic symbols not containing ´> ´

1.1.4.2. Example

```
´give your choice please´
```

1.1.4.3. Semantics

The string quotes (´ ´) are introduced to allow the handling of arbitrary sequences of symbols. Strings may be stored and be processed in ´string´ variables.

1.2. Variables

1.2.1. Syntax

```
<variable> ::= <class_var>
             |<var_items>

<class_var> ::= <hierar_variable>
              |<non_hierar_variable>

<hierar_variable> ::= <non_hierar_variable>.<class_var>

<non_hierar_variable> ::= <name>
                        |<subscripted_variable>

<subscripted_variable> ::= <name>[<sub_index>]

<sub_index> ::= <arithmetic_expression>,<arithmetic_expression>,
                    <arithmetic_expression>
              |<arithmetic_expression>,<arithmetic_expression>
              |<arithmetic_expression>

<var_items> ::= (<name>).<class_var>
              |(<name>).FILE
              |(<name>).TYPE
```

1.2.2. Examples

```
CUSTOMER
CUSIT.NAME
(CUS).NAME[1]
TAB[I+J,20]
(CUS).FILE
```

1.2.3. Semantics

A variable denotes a memory space wherein a single value or a sequence of values is kept. This data reference is used in expressions to get or calculate new values. The value of a variable may be changed at will by means of assign statements (see section 1.6.2.) The type of the value(s) of a particular variable is defined by the declaration of this variable itself or by the declaration of the containing structure, like an array or a database record type (see section 1.5.).

1.2.3.1. Subscripts

Subscripted variables designate values which are components of one, two or three-dimensional arrays. Each arithmetic expression of the subscript list denotes one subscript of the subscripted variable. The complete subscript expression is enclosed in square brackets. The array component referred to by a subscripted variable is specified by the actual numerical value of its subscripts.

Each subscript has to denote an integer value. So it can be an adding or multiplying expression of unsigned integers and 'integer' variables or, more simply, one unsigned integer or one variable of type integer. Furthermore, the value of the subscripted variable is defined only if the value of each subscript is within the declared bounds of the array.

1.2.3.2. Hierarchy

Hierarchical variables designate values which are components of group variables. In fact they allow one to address one field of some complex data structure. A field selection is always denoted by a non-hierarchical variable. Two consecutive field selections are separated by a period which represents a jump to a lower level in the structure. The left most field selection of a hierarchical variable gives the name of the group variable. The right most field selection gives the addressed part of the structure; its declaration determines the type of the value.

1.2.3.3. Items variables

These variable expressions designate the data item values of some record stored in the database. The name enclosed between the parentheses denotes the name of some record variable (that's to say a variable containing the reference of some database record). The whole expression "(‹name›)" represents the call of a function which automatically creates a data structure containing all the data item values of the record which is referenced by the record variable. So the expression is quite dynamic: it first verifies the existence of the reference variable, then it generates a structured data record with the data item values of the database record and finally it addresses the indicated component of the structure. It follows that, once the data structure is generated, the designation of a component is exactly similar to the principle of a hierarchical variable. Of course the field selection sequence must denote decomposable data items defined for the considered record type. Only the right most field selection denotes an elementary data item.

There are two particular field selections: FILE and TYPE. These two variable expressions make it possible to address the file name or the type name of the record that is referenced by the record variable.

Finally it is possible to designate the whole generated items structure of a record: ‹class_var› has only to denote the character string 'ITEMS'. This data structure reference can only be used to explicitly

assign the data item values of the referenced record to some variable that has been defined to receive the item values of an instance of the given record type (variable declared ‹items_of›, see section 1.5.).

1.3. Expressions

1.3.1. Collection expression

1.3.1.1. Syntax

```
<collection_expression> ::= <range> | <DB_object_set>

<range> ::= <range_expr>..<range_expr>

<range_expr> ::= <unsigned_integer> | <name>

<DB_object_set> ::= <name> <predicate>

<predicate> ::= (<coll_cond_term>) and (<predicate>)
              |<coll_cond_term>

<coll_cond_term> ::= not<coll_cond_term>
                   |<coll_cond_primary>

<coll_cond_primary> ::= <relation_condition>
                      |()
                      |(<predicate>)

<relation_condition> ::= <relation_operator> <name>)
                       |<relation_operator> <belonging_cond>)

<relation_operator> ::= (<name>:
                      |( :

<belonging_cond> ::= <name> <test_operator> <arithmetic_expression>
```

1.3.1.2. Semantics

A collection expression always designates a sequence (an ordered set) of objects: integer values (range) or database records (DB object set).

1.3.1.2.1. Range

A range denotes the sequence of values up from the minimum value (left range expression) to the maximum value (right range expression). The two expressions represent the bounds of the range. Each boundary must be denoted by an unsigned integer or by a simple 'integer' variable.

Examples:

2..5
I..J
1..K

1.3.1.2.2. DB object set

A DB object set is defined by a name followed by a predicate. The name must denote a predefined set of database objects. More precisely, the name either designates some record type of the database (it denotes all the instances of the record type) or the general string 'RECORD' (it denotes all the records of the database independently of their type). The predicate will act like a filter on the sequence of records denoted by the name. Hence it defines a new DB object set which is a subset of the set defined by the name.

The predicate is an expression which may contain several conditions. These conditions are called collection condition primaries. It is possible to combine the collection condition primaries with the binary logical operator 'and'. The unary operator 'not' is allowed, too. In the case of combination of collection condition primaries, it is required that this sequence of conditions is enclosed between parentheses (<predicate>). But the predicate may also be empty, meaning that no further filter is imposed on the records to be selected.

For the effective ADL, the selection realized by means of the predicate may only use one of the basic access mechanisms provided by the GAM. In other words, the predicate can only denote some access path condition (relation condition) and/or some access key condition (belonging condition). A record of the sequence denoted by the name will be selected if it satisfies the association condition to some origin record via the given access path type and/or if its associated data item values satisfy the belonging condition to some defined value(s) of the access key.

Relation condition.

The objects of the database are related to each other. In particular a record of some type may be linked through a certain relation to a set of objects. This set of targets is identified by the record and the relation. The set is really a sequence as the elements of the set are ordered.

The relation condition uses the concept of set of targets. In effective ADL algorithms this target set may only be a sequence of records, as the relationship must correspond to some access path type. Therefore the relation operator must denote either the name of an access path type defined between two record types or no name at all. In this case it designates the nameless access path type existing between a record type and its data items.

If the relation condition concerns an access path type between two record types, the expression following the relation operator may only specify the name of a record variable which references the origin record. If the relation condition denotes the nameless access path type between a record type and its data items, the condition must specify some belonging condition.

Belonging condition

Checking the value of some object with respect to the value of the arithmetic expression , the belonging condition may only apply a selection criterion on data item values. In addition, as the access condition must correspond to a simple access by key, the name of the belonging condition can only denote an elementary data item which is a component of the access key. The evaluation of the arithmetic expression during run time has to give one single value. It cannot denote another DB object set although the syntax allows it (see section 1.3.2.).

If the access key is decomposable, a relation condition must be stated for each of its components, giving an 'and' combination of collection condition primaries. The semantics imposes that the constituent data items are present in only one collection condition primary and that they are specified in the order they are defined to form the access key.

Moreover, as an effective ADL program has to contain only clauses which correspond directly to the functions of the access module, it becomes obvious that the syntax is too general in admitting the logical negation operator ('not'). Semantically a negated collection condition primary must be rejected.

Examples:

CUSTOMER(C_OR:ORD) defines the sequence of records of type CUSTOMER which are linked via the access path type C_OR to the record of type ORDER referenced by ORD.

PRODUCT(:NB_PRO=123) defines the sequence of records of type PRODUCT which contain a value for the data item NB_PRO that matches the access key value 123.

RECORD(WORKS:DEP) if we assume that the access path type WORKS links the record type DEPARTMENT to the two record types EMPLOYEE and WORKER, the DB object set defines the sequence of records of type EMPLOYEE or WORKER which are related via the access path type WORKS to the record DEPARTMENT which is referenced by the variable DEP.

Remark:

If the name of the DB object set denotes the string 'RECORD', the type of the selected records is unknown. As an access key is always defined for one record type, it follows that an access by key, and thus a predicate specifying a belonging condition, has to be excluded. At the same time we exclude the possibility of an empty predicate, as this would mean an access to all the records of the database. This restriction is put forward to avoid an abusive use of general record variables. They should only be used for the access within a multiple target access path type.

1.3.2. Arithmetic expression

1.3.2.1. Syntax

```
<arithmetic_expression> ::= <factor> <adding> <arithmetic_expression>
                          |<adding> <arithmetic_expression>
                          |<factor>

<factor> ::= <term> <multiplying> <factor>
           |<term>

<term> ::= <primary> ^ <primary>
         |<primary>

<primary> ::= <string>
            |<unsigned_number>
            |<logical_value>
            |<variable>
            |<call_st>
            |<nullref>
            |<DB_object_set>
            |(<arithmetic_expression>)

<unsigned_number> ::= <unsigned_integer>
                    |<unsigned_real>

<adding> ::= + | -

<multiplying> ::= * | /

<nullref> ::= ( )
```

Examples:

```
A+B-(A/B)
SIN(A*B)
IT.NAME[I]
CUSTOMER(:NB_CUS=111)
```

1.3.2.3. Semantics

An arithmetic expression represents a formula for processing values. The value of the expression is obtained by executing the indicated operations on the actual values of the primaries appearing in the expression. However the arithmetic operators may only be applied on arguments having a numeric value. Hence an arithmetic expression specifying one or the other arithmetic operation can only contain unsigned numbers, variables declared of type integer, real or numeric (see section 1.5.) and numeric functions. Therefore if the actual value of a primary is non-

numeric, it forms the only primary of the expression.

The actual value of a primary is obvious in the case of an unsigned number, a string constant, a logical constant and nullref ('null' reference). For a variable, it is its current value. The class of possible values is delimited by the variable declaration. For a function identifier, it is the value which is calculated by the computing rules of the function when applied to the current values of the function parameters given in the expression. The type of this value is determined by the function definition stored in the workbench database (see MODULE in chapter 4). For the designation of a DB object set, it is the sequence of records verifying the selection condition expressed by the name and the predicate (see section 1.3.1.2.2.). This type of arithmetic expression may only be used in an assign expression to express the access to a single identified record of the database (see section 1.6.2.).

Finally the value of an arithmetic expression enclosed in parentheses is determined by recursively analysing the constituent primaries of this expression. The parentheses represent a way to change the usual precedence of the operators:

- first : ^
- second : * /
- third : + -

1.3.3. General expression

1.3.3.1. Syntax

```
<general_expression> ::= <general_factor> or <general_expression>
                         |<general_factor>

<general_factor> ::= <general_term> and <general_factor>
                     |<general_term>

<general_term> ::= not<general_primary>
                   |<general_primary>

<general_primary> ::= <test_expression>
                      |(<general_expression>)

<test_expression> ::= <arithmetic_expression> <test_operator>
                      <arithmetic_expression>
```

Examples:

```
(A+B<C)
FOUN=TRUE and I<MAX
(PRO).Q_STK <= 10
```

1.3.3.3. Semantics

A general expression represents one test expression or a logical combination of several of these boolean expressions. The logical operators are the binary operators "or", "and" and the unary operator "not". The definition of the syntax expresses the precedence of these operators:

- first : not
- second : and
- third : or

The possibility of a general expression enclosed in parentheses provides the opportunity of changing this fixed precedence.

A test expression itself is formed of two arithmetic expressions. The evaluation of each expression has to give only one value and the two values being compared have to be of comparable types. It follows that the arithmetic expression may denote neither a variable of some complex data type nor a DB object set. In the case where two boolean values or two record variables form the test expression, the test operator can only be the identity of the difference sign. This restriction also applies to the comparison of a record variable to the null reference ("()"). As the evaluation of each arithmetic expression has to be done before the result of the test expression can be determined, it is obvious that the arithmetic operators have a higher precedence than the test operators.

1.4. Global Algorithm Structure

1.4.1. Syntax

‹algorithm_structure› ::= ‹algorithm_heading› ‹algorithm_body› .

‹algorithm_heading› ::= algorithm ‹name›

‹algorithm_body› ::= ‹intern_declaration_part› ‹statement_part›

1.4.2. Semantics

The global structure of an ADL algorithm is actually divided in three parts:

- the algorithm heading: that is the algorithm name. This name corresponds to a module listed in the conceptual schema of the workbench database (see chapter 4).

  Examples.

  ```
  algorithm CALCULSIN
  algorithm TRAITEMENT_CLIENT
  ```

- the intern declaration part: it is the first part of the algorithm body, which will be explained in the next section;

- the statement part: the second part of the algorithm body will also be explained later.

1.5. Declaration Part

1.5.1. Syntax

```
<intern_declaration_part> ::= <type_declaration_part>
                              <variable_declaration_part>

<type_declaration_part> ::= type <type_declarations>
                          | <empty>

<type_declarations> ::= <type_declaration>
                      | <type_declaration>;<type_declarations>

<type_declaration> ::= <name> = <variable_type>

<variable_declaration_part> ::= var <variable_declarations>
                              | <empty>

<variable_declarations> ::= <variable_declaration>
                          | <variable_declaration>;
                                <variable_declarations>

<variable_declaration> ::= <names> : <variable_type>

<names> ::= <name>
          | <name>,<names>

<variable_type> ::= <simple>
                  | <structured>
                  | <name>
```

1.5.2. Semantics

The intern declaration part consists of two parts. First, we have the type declaration part and then the variable declaration part.

Each of the two declaration parts can be empty.

If the type declaration is not empty, it gives a list of declarations. Each declaration associates a specific name or identifier with the corresponding data type specified in the second part of the declaration. The declared data type represents a non standard type which is described in terms of the data types offered by ADL (see next section) or in terms of other non standard data types. These data types have to be defined beforehand in the type declaration. The principle of the type declarations is therefore the same as in the programming language Pascal.

If the declaration part is not empty, it gives a list of declarations; each of these associates one or more variable names (or identifiers) with the data type indicated in the right part of the declaration. This declaration determines the set of possible values which may be assigned to the given variable(s).

There are three kinds of variable types: simple, structured, name.

1.5.2.1. Data types

1.5.2.1.1. Syntax

```
<variable_type> ::= <simple>
                  | <structured>
                  | <name>

<simple> ::= boolean
           | real
           | integer
           | numeric(<unsigned_integer>,<unsigned_integer>)
           | string(<unsigned_integer>)

<structured> ::= <group>
               | <array>
               | <items_of>
               | <ref_of>

<group> ::= group <field_list> end

<field_list> ::= <variable_declaration>
               | <variable_declaration>;<field_list>

<array> ::= array [<index_lst>] of <component_type>

<index_lst> ::= <index>,<index>,<index>
              | <index>,<index>
              | <index>

<index> ::= <unsigned_integer>

<component_type> ::= <simple>
                   | <ref_of>
                   | <items_of>
                   | <name>

<items_of> ::= items_of <name>

<ref_of> ::= ref of <name>
           | ref of RECORD
```

1.5.2.1.2. Examples

```
type ENTIER = integer;
     ADRESSE = group
                 RUE : string(30);
                 NUM : ENTIER;
                 VILLE : string(30);
                 CODE : numeric(4,0)
               end;
```

```
     TABLE = array [10] of integer

var  FOUND, PRESENT : boolean;
     ADR_CUS, ADR_DEAL : ADRESSE;
     FACTURE : group
                 NUM_FACT : ENTIER;
                 TOTAL : numeric(10,2)
               end;
     CUS : ref of CUSTOMER;
     CUS_REC : items_of CUSTOMER
```

1.5.2.1.3. Semantics

-> Simple variable types.

The first kind of data variable type, we have, is the simple data type.

The simple data variables contain only one information:

- boolean variables assume two values: true and false;
- integer variables assume positive or negative integer values and zero;
- real variables assume positive or negative values and zero;
- numeric variables assume negative or positive values with a specified number of digits in the integer part and with a specified number of digits in the decimal part, plus zero; this is a special case of the real data type;
- string variables assume a specified number of characters.

-> Structured variable types.

The second type of variable type is the structured type. In this case, the number of informations given for each of these types can be more than one information.

1. Group.

The group variable data type is the first structured type. It corresponds to the record data type in Pascal.

A group is a structured data in which the different elements (specified in the field list by means of standard variable declarations) can have differents types. Each component corresponds to one field.

2. Array.

An array declaration declares one or several identifiers representing arrays of subscripted variables, it gives the dimension of the arrays, and their bounds plus the type of the elements of the array.

There are three types of arrays: one-dimensional arrays, two-dimensional arrays and three-dimensional arrays. We had to restrict the effective ADL language to these three types of array because of the limitations we meet in the Cobol language. It does not admit more than three dimensional arrays.

The numbers specified in the indices are the maximum number of elements the array can contain. An array is, thus, a collection of identical elements whose type is specified in the component type clause. Here, we have another restriction: the component type cannot be a structured type.

3. Items_of.

The items_of variables serve to keep instances of database record types. Thus, the name specified in the declaration must denote some record type of the database; it indicates the type of the records the variable is allowed to contain.

4. Ref of.

The ref_of variables serve to keep references to database records. The implicit type of the variable is integer (see section 1.8. of chapter 2). If a name is specified, it must be a record type of the database; the variable declared is thus a reference to records that have that type. If the name RECORD is specified, it means that the variable declared is a reference to records of the database of different unknown types. The variable can be used to refer all records of the database.

-> Name variable types.

In this case, the variable type is to be found in the type declarations.

This last kind of variable declaration does not describe in details the type of the variable but specifies a name referring to a non standard data type previously declared in the declaration part. Thus, those declarations define one or more variables to be of the type given in the type declaration part.

1.6. Statement Part

1.6.1. Structure of the statement part

1.6.1.1. Syntax

```
<statement_part> ::= begin <statements> end

<statements> ::= <statement>
                 | <statement> ; <statements>

<statement> ::= <ass_st>
                | <for_st>
                | <if_st>
                | <while_st>
                | <db_mod>
                | <next_st>
                | <exit_st>
                | <call_st>
                | <return_st>
                | <dummy>

<dummy> ::= <empty>
```

1.6.1.2. Examples

```
begin
     I := 0;
     while I<>N do
         I:=I+1;
         PRINT!(I)
     endwhile;
end
```

1.6.1.3. Semantics

The statement part of an ADL program is a set of statements preceded with "begin" and terminated by the keyword "end".

1.6.2. Assignment statement

1.6.2.1. Syntax

```
<ass_st> ::= <variable> := <assign_expression>

<assign_expression> ::= <arithmetic_expression>
                      | <general_expression>
```

1.6.2.2. Examples

```
FOUND := true;
(CUS).ADRESSE := 'sunset boulevard San Francisco';
PRESENT := (X>TAB[I]) and (X<TAB[J]);
TEMP := REC1.FIELD1 + REC2.FIELD2;
CUS := ( )
CUS := CUSTOMER( :NBCUS = 123)
```

1.6.2.3. Semantics

Assignment statements serve to assign a certain value (right part) to a variable (left part). The type of the right part of the assignment statement, which can be expressed explicitly (value) or implicitly (variable), and the type of the assigned variable must be the same.

In the particular case of an assignment statement with a general expression, the value of the right part of this assignation is a logical value; so the type of the variable in the left part must be boolean.

Semantically, the assignment statement may only apply to variable of a simple type (real, integer, numeric, string, boolean). The only exception is the specification of a DB object set in the assign expression. In this case, the statement represents an access to a single identified record of the database. In other words, the DB object set (see section 1.3.1.2.2.) must denote a sequence of one or zero records.

1.6.3. For statement

1.6.3.1. Syntax

```
<for_st> ::= for <variable> := <collection_expression> do
                <statements> endfor
```

1.6.3.2. Examples

```
for CUS := CUSTOMER( :NAME = 'DUPONT' ) do
        TR_CUS!(CUS)
endfor

for I:=1..10 do
        TAB[I] := EXP!(I)
endfor
```

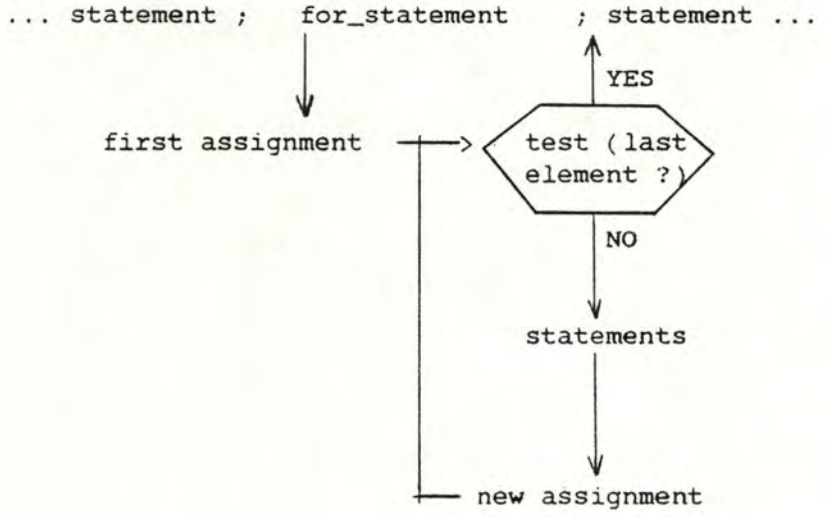
1.6.3.3. Semantics

The for statement is a way to represent an enumerative loop. The collection expression (see section 1.3.1.) may have quite different meanings, depending whether it denotes a range or a DB object set. Generally speaking, the for statement allows one to specify a process to be executed on each element of the defined sequence:

- a range defines a sequence of integer values;
- a DB object set defines a sequence of database records.

At each iteration, the value of the reference of the sequence element to be processed by the loop is assigned to the control variable and a test is performed to check if the assignment just done was not the last. If the end of the sequence is encountered, the statement following the for statement is executed.

This little process can be described as this :



It is obvious that the type of the control variable must be integer in the case where the collection expression denotes a range; otherwise the loop variable must denote a record variable defined to refer the records of the selected sequence.

1.6.4. Next statement

1.6.4.1. Syntax

```
<next_st> ::= next
            | next <name>
```

1.6.4.2. Example

```
next CUS
```

1.6.4.3. Semantics

The next statement forces the termination of the current execution of the statements in a for loop. If a name is specified, this name must be the name of a loop control variable. In that case, the termination concerns the loop controled by this variable. The termination of the loop implies the abortion of all the loops contained in it.

If no name is given, that means that the abortion concerns the inmost loop body.

A new assignment is performed for the loop concerned with the abortion.

1.6.5. Exit statement

1.6.5.1. Syntax

```
<exit_st> ::= exit
            | exit <name>
```

1.6.5.2. Example

```
exit PROD
```

1.6.5.3. Semantics

The exit statement allows one to abort the execution of a loop and to execute the statement following the end of the for statement.

If a name is specified, it must be the name of a control variable. The abortion then concerns the loop controled by this variable and of course the loops inside this one.

Otherwise with no name specified, the abortion concerns the inmost loop.

1.6.6. If statement

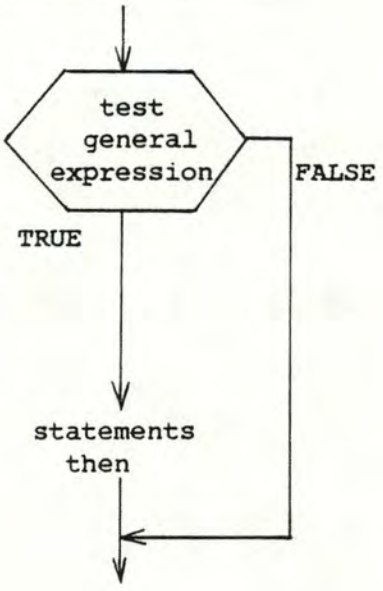
1.6.6.1. Syntax

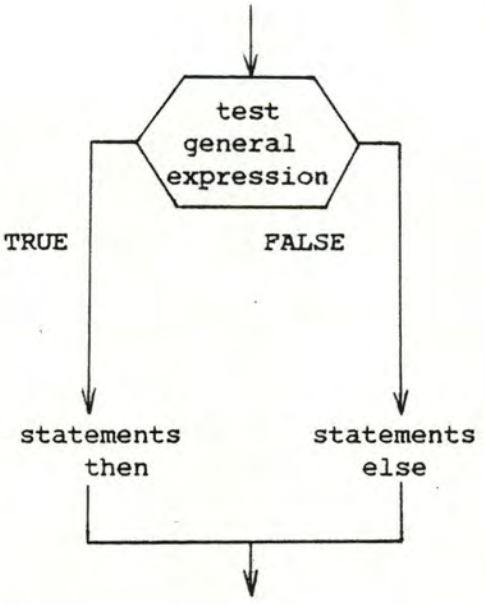
```
<if_st> ::= if <general_expression> then <statements> endif
          | if <general_expression> then <statements>
            else <statements> endif
```

1.6.6.2. Examples

```
if (TIME = 10PM) and (NB_PERSON < 10)
        then PRINT!(MESS_1)
        else PRINT!(MESS_2)
endif

if TAB[I] = X
        then
            if TAB[J] = Y
                then  PRINT!(MESS_OK)
            endif
        else
            PRINT!(MESS_NOT_OK)
endif
```

1.6.6.3. Semantics





These two if statements cause certain sequences of statements to be executed and others to be skipped, depending on the value of the general expression (as shown on the figure above).

The type of the general expression is boolean. The statements following the "then" clause will be executed if the value of the general expression is true. They will not be executed if the value is false; in this case the statements following the "else" clause will be executed, if any.

The endif clause always refers to the last if encountered.

1.6.7. While statement

1.6.7.1. Syntax

```
<while_st> ::= while <general_expression> do <statements> endwhile
```

1.6.7.2. Example

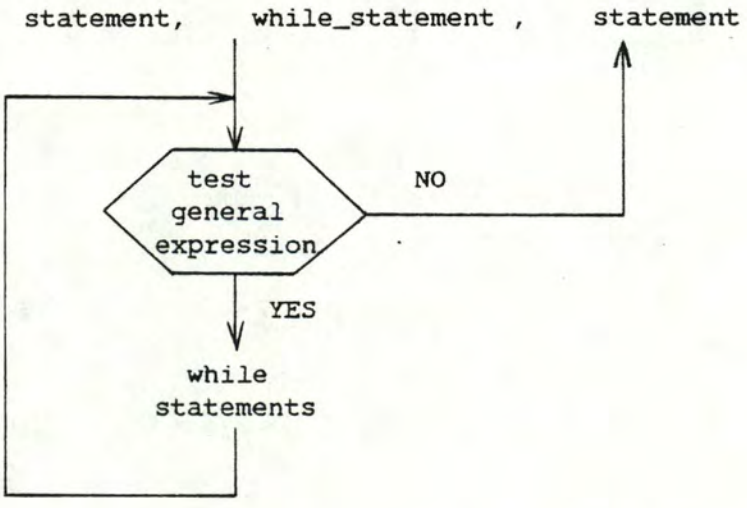
```
while (A<>B) and (PRESENT = false) do
        I:=I+1;
        if TAB[I] = X then A:=X;
                           PRESENT := true
                    endif
endwhile
```

```
while (I<N) do
        TR_EXP!(I);
        I:=I+1
endwhile
```

1.6.7.3. Semantics

Like the for loop, the while loop causes the iterative execution of a sequence of statements but it does not perform any assignment to a loop control variable.

This process can be described as follows:



The test is performed on the value of the general expression; if the evaluation of that expression gives as result true then the statements are executed. Otherwise, the statement following the while statement is executed. It is obvious that the type of the general expression is boolean.

1.6.8. Call statement

1.6.8.1. Syntax

```
<call_st> ::= <proc_name>
            | <proc_name> (<param_list>)

<param_list> ::= <variable>
               | <variable>,<param_list>
```

1.6.8.2. Examples

```
SIN!(A,SINA)
SHOWTIME!
```

1.6.8.3. Semantics

The call statement allows one to invoke a procedure or a function. The functions and procedures invoked in an ADL program must be listed in the conceptual schema of the workbench database. Every function corresponds to a separate module (see chapter 4). The identifier name of a called function or procedure must always end with the special character "!". The call statement allows the passage of arguments to the called function or procedure.

1.6.9. Return statement

1.6.9.1. Syntax

```
<return_st> ::= return
              | return (<param_list>)

<param_list> ::= <variable>
               | <variable>,<param_list>
```

1.6.9.2. Examples

```
return;
return (FOUND,ADRESSE);
```

1.6.9.3. Semantics

In an invoked procedure (or function), the return statement orders the come back to the main program. It allows the passage of arguments (as results) to the calling program.

1.6.10. Data base modification statement

1.6.10.1. Syntax

```
<db_mod> ::= <modif>
           | <creat>
           | <del>
```

1.6.10.2. Semantics

These statements modify a record of the database the ADL program is defined to work on.

1.6.11. Create statement

1.6.11.1. Syntax

```
<creat> ::= create <name> := <name> <creat_conds>

<creat_conds> ::= <creat_cond>
                | (<conditions>)

<conditions> ::= <creat_cond>
               | <creat_cond> and <conditions>

<creat_cond> ::= <attach_cond>
               | <item_cond>

<item_cond> ::= <relation_operator>
                <name> = <arithmetic_expression> )

<attach_cond> ::= <relation_operator> <name> )

<relation_operator> ::= (<name>:
                      | ( :
```

1.6.11.2. Example

```
create COM:=COMMANDE((CC:CLI) and (:DATE=DATE_JOUR) and
                     (:NCOM=DERNIER_NUMCOM + 1))
```

1.6.11.3. Semantics

A instance of the record type denoted by the name given in the right part of the assignment is created. The reference to this record is assigned to the record variable named in the left part of the assignment. The record created will verify the create condition(s). In other words, the created record will be associated to the data item values specified in the item conditions and it will be linked to the origin records via the corresponding access path type, both given in the attach condition.

1.6.12. Modify statement

1.6.12.1. Syntax

```
<modif> ::= modify <name> <modif_conds>

<modif_conds> ::= <attach_cond>
                | <items_conds>
                | <detach_cond>

<items_conds> ::= <item_cond>
                | ( <conds> )

<conds> ::= <item_cond>
          | <item_cond> and <conds>

<item_cond> ::= <relation_operator>
                      <name> = <aritmetic_expression> )

<attach_cond> ::= <relation_operator> <name> )

<detach_cond> ::= <relation_operator> <unsigned_integer> <name> )
```

1.6.12.2. Examples

```
modify CLI(( :ADR='NAMUR' )and( :TEL=081818181));
modify COM(CC:0 CLI)
modify COM(CC:CLI)
```

1.6.12.3. Semantics

The name in the modify statement must denote a record variable, that's to say a reference to a database record type. The record implied in the modify statement, will be modified in the following way:

- <items_conds> condition: the values of the data items of the record will be modified so that the value of each data item will be replaced by the result of the evaluation of the arithmetic expression;
- <attach_cond> condition: the record of the modify statement will be attached to the record indicated by the name specified in the attach condition via the access path type specified in the relation operator;
- <detach_cond> condition: the record of the modify statement will be detached from the record referenced by the name of the detach condition, it was linked to via the access path type specified in the relation operator. The unsigned integer is always 0 so that we can read the statement this way: modify the record (whose reference is given in the statement) so that it is linked to no record (specified in the condition) via the specified access path type.

1.6.12.4. Semantics of the conditions

For an item condition, the name in the condition must be a data item of the considered record type. The type of that item must be the same as the type of the result of the evaluation of the arithmetic expression. For an attach and a detach condition, the name specified must denote a record variable; and the name specified in the relation operator must denote an access path type defined between the two database record types which are referenced in the modify statement (for more details see section 4.2.2. of chapter 2).

1.6.13. Delete statement

1.6.13.1. Syntax

<del_st> ::= delete <name>

1.6.13.2. Example

delete CLI

1.6.13.3. Semantics

The name specified in the delete statement must denote a variable defined to contain a reference of some record of a certain type.

The record specified is deleted, and so are all the records which are mandatory targets of this record by some access path type (recursive effect).

2. EXTENSIONS TO THE EFFECTIVE ADL DEFINITION

This part of the chapter shortly presents some extensions which have been defined to get a first syntactic definition of the whole ADL language. To distinguish clearly between the two language definitions, we shall call this extended ADL grammar 'predicative' ADL. We are well aware that the proposed syntax is not general enough to allow all the expressions the ADL should offer. But we think that these first ideas may be helpful in a later and definite specification of the complete ADL grammar.

The two terms "effective" and "predicative" ADL must not be understood as if they designate two completely different languages. As a matter of fact, the effective ADL represents a subset of the predicative ADL.

In this part we shall describe the extensions by using the same representation structure as in the first part. For each section of the first part, we give the extensions which have been implemented for the syntactical analyser.

2.1. Basic Symbols, Names, Numbers and Strings

The inclusion of a new data type (the lists) and the generalization of the expressions and the for statement require the introduction of new keywords (one declarator and three separators) as well as new brackets (brace brackets) and test operators (in, not_in). So the extension only concerns the delimiters of the basic symbols (see section 1.1.1.4.)

2.1.1. Syntax

‹test_operator› ::= ‹ | › | ‹= | ›= | ‹› | = | in | not_in

‹separator› ::= , | . | ; | := | .. | // | do | then | else | order | if_no | endfor
| endif | endwhile

‹bracket› ::= (|) | [|] | { | } | begin | end

‹declarator› ::= algorithm | type | var | group | array | list | of | items_of
| ref | boolean | string | real | integer | numeric

2.2. Variables

2.2.1. Syntax

```
‹variable› ::= ‹class_var›
             | ‹var_items›
             | ‹name›//‹name›
```

2.2.2. Example

```
CUSTOMER//CUS
```

2.2.3. Semantics

The syntax of a variable has been extended to allow the general set expression of the Data Designation Language (see section 2.2. of chapter 2). The first name denotes a database record type and the second is the name of a record variable. Considering that a DB object set is henceforth formed of a variable and a predicate (see section 2.3.1.2.3.), this allows us to write the selection expression of a sequence of database records and to define the record variable as some label of this records sequence.

Note that contrary to the DDL expression, the symbol separating the two identifiers is a double slash character. In fact, the character had to be doubled to avoid an ambiguity with the dividing symbol of the arithmetic expression.

2.3. Expressions

In the predicative ADL syntax, the collection expression is substantially different; the arithmetic expression reveals a small modification and the general expression is not changed at all.

2.3.1. Collection Expression

2.3.1.1. Syntax

```
‹collection_expression› ::= ‹range›
                          | ‹list›
                          | ‹DB_object_set›
```

```
<range> ::= <arithmetic_expression>..<arithmetic_expression>
          |*..<arithmetic_expression>
          |<arithmetic_expression>..*
          |*..*

<list> ::= {<list_enumeration>}

<list_enumeration> ::= <list_element>,<list_enumeration>
                     |<list_element>

<list_element> ::= <assign_expression>

<DB_object_set> ::= <variable> <predicate>

<predicate> ::= (<coll_cond_factor>) or (<predicate>)
              |<coll_cond_factor>

<coll_cond_factor> ::= (<coll_cond_term>) and
                                    (<coll_cond_factor>)
                     |<coll_cond_term>

<coll_cond_term> ::= not<coll_cond_primary>
                   |<coll_cond_primary>

<coll_cond_primary> ::= <relation_condition>
                      |()
                      |(<predicate>)

<relation_condition> ::= <relation_operator> <co> <variable>)
                       |<relation_operator> <co>
                                    <DB_object_set>)
                       |<relation_operator> <co>
                                    <belonging_cond>)

<relation_operator> ::= (<name>:
                      |( :

<co> ::= [<cardinal>] | <ordinal> | <empty>

<cardinal> ::= <range> | * | <unsigned_integer> | <name>

<ordinal> ::= #<cardinal>

<belonging_cond> ::= <variable> <test_operator>
                                <arithmetic_expression>
```

2.3.1.2. Semantics

Here the collection expression designates an ordered set of simple value expressions (range and list) or a sequence of data item values or database records (DB object set). By simple value expression we mean an assign or arithmetic expression whose evaluation produces one single value.

2.3.1.2.1. Range

The semantics of the range expression is exactly the same apart from the fact that the expression giving the values of the lower and upper bound can be more complex. They must nevertheless define a simple value expression. The symbol '*' used as a maximum value means infinity, while '*' used as a minimum value means minus infinity.

Examples:

```
I+J..TAB[K]
1..*
```

2.3.1.2.2. List

A list is formed of a sequence of simple value expressions. Each of these expressions denotes one element of the list. Two consecutive expressions are separated by a comma. The complete set of elements is enclosed between '{}'.

Examples:

```
{I,I+J,I*J,I/J}
{'IKO','IK','I','KO','K','O'}
```

2.3.1.2.3. DB object set

In the predicative ADL, a DB object set is formed of a variable identifier followed by a predicate. The variable must denote a predefined set of DB objects and the predicate again acts like a filter on this ordered set of DB objects. The fundamental difference is that the syntax of a variable (contrary to a simple name) allows the specification of some subscript as well as the assignation of some record variable in the expression of the DB object itself. Moreover the expression of the predicate is considerably generalized as it may specify any kind of access condition, possibly combined with a cardinal or ordinal condition. In other words, the selection condition must no longer correspond to some elemtary access methods provided by the GAM (access by key, access within an access path); it allows any condition expression of the Data Designation Language (see section 2 of chapter 2).

More precisely , the DDL requires that the variable designates a predefined set of records or data item values: it must denote the name of some record type of the database, the string 'RECORD' or the name of some data item. In addition, it may specify the name of some record variable which is separated from the record type name by '//', in order to notify that from there on, the use of this label will designate this set of selected records (see section 2.2.). Finally the variable may include a subscript. This subscript does not represent a simple index of some array as, in a predicative ADL algorithm, the exact order of the set of records

is defined but not the effective implementation of this sequence. It just means that the selected object(s) must be the i-th element(s) in their association to the origin DB object with respect to some indicated order or to some default order (natural ordering).

The predicate is again an expression of several conditions. The only difference is that the collection condition primaries may also be linked by the binary operator 'or'. And there is no restriction on the use of the negation operator ('not'). Each collection condition primary is formed of a relation condition which specifies the association to some DB records or data items referenced by a variable or DB object set expression. The relation condition may also include a belonging condition.

Relation condition

In a predicative ADL program, the relation condition may specify an association condition between records (access path type), between a record and its data item values or between data item values (decomposition of a decomposable data item). In the last two cases, the relation condition normally includes a belonging condition and must not specify any name in the relation operator.

Moreover the relation condition may contain a cardinal or ordinal relation criterion. Both syntactic structures define one integer value (i) or an interval of integers ([i..j] or #i..j). To explain their exact meaning, we shall call original set the sequence of DB objects defined by the variable and reference set the sequence of objects denoted by the variable, DB object set or belonging condition of the relation condition.

- Cardinal: the criterion is true for an instance of the original set if it is related to 'k' elements of the reference set and 'k' belongs to the sequence of integers defined by the cardinal expression (i<=k<=j).

- Ordinal: the criterion is true for an instance of the original set if:
  - it is related to 'k' DB objects
  - 'k' is greater or equal to the minimum value of the range specified by the ordinal expression (k>=i)
  - all the objects related to this instance which have a rank between i and min(k,j) belong to the reference set.

Finally note that the reference set may be denoted by a record variable or a record type name if it is a sequence of records. If the reference set does not denote any predefined set of records or data item values, it is represented by yet another DB object set, as a predicate expression is necessary to determine this particular set of DB objects. The number of overlapping DB object set expressions is not limited.

Belonging condition

The belonging condition has been extended in that it allows the specification of a condition on any data item of a record; it has no longer to be a component of an access or order key. Furthermore two test operators have been added (in, not_in: see section 2.1.), which make it possible to express a belonging condition on records with respect to the reference set denoted by the arithmetic expression.

Examples:

ORDER(OR_OL:[3..4]ORDER_LINE) defines the set of records of type ORDER which are linked via the access path type OR_OL to three or four records of type ORDER_LINE

ORDER(OR_OL:#3..* ORDER_LINE(:Q>10)) defines the set of records of type ORDER which are linked to three or more records of type ORDER_LINE so that those from rank three up to the last contain a data item value Q greater than 10.

2.3.2. Arithmetic Expression

The definition of this expression remains the same except that in the syntax rule of the primary, the non terminal ‹DB_object_set› has been replaced by the more general non terminal ‹collection_expression› (see section 2.3.1.)

2.3.2.1. Syntax

```
‹primary› ::= ‹string›
            | ‹unsigned_number›
            | ‹logical_value›
            | ‹variable›
            | ‹call_st›
            | ‹nullref›
            | ‹collection_expression›
            | ( ‹arithmetic_expression› )
```

2.3.2.2. Semantics

This substitution means that in the predicative ADL, a list or a range expression may be used in test expressions or in assign expressions. These types of operations are no longer restricted to simple data types.

2.3.3. General Expression

There has been no change whatsoever as far as this syntactic structure is concerned.

2.4. Global Algorithm Structure

There has been no change in the general structure of the algorithm.

2.5. Declaration Part

The main changes we have here are some generalizations for the arrays declarations and the lists declarations.

2.5.1. Array declaration.

The dimension of an array is no longer limited like in the effective ADL. Now, the number of dimensions is not restricted.

Syntax

```
<array> ::= array [<index_lst>] of <component_type>

<index_lst> ::= <index>
              | <index>,<index_lst>

<index> ::= <unsigned_integer>

<component_type> ::= <simple>
                   | <items_of>
                   | <ref_of>
                   | <name>
```

2.5.2. List Declaration

The extension of the collection expression to lists requires the inclusion of list declarations.

2.5.2.1. Syntax

```
<list_of> ::= list of <component_type>
            | list [<unsigned_integer>] of <component_type>
```

2.5.2.2. Semantics

We can declare a list in two different ways. First, we can declare a list without a maximum bound (for example: list of integer). The maximum bound is defined somewhere in the system by the designer but not in an ADL program. Then, we can define a list with a well defined size (for example: list [10] of integer).

2.6. Statement Part

The main changes made in the statement part concern the for statement and the delete statement.

2.6.1. For statement

2.6.1.1. Syntax

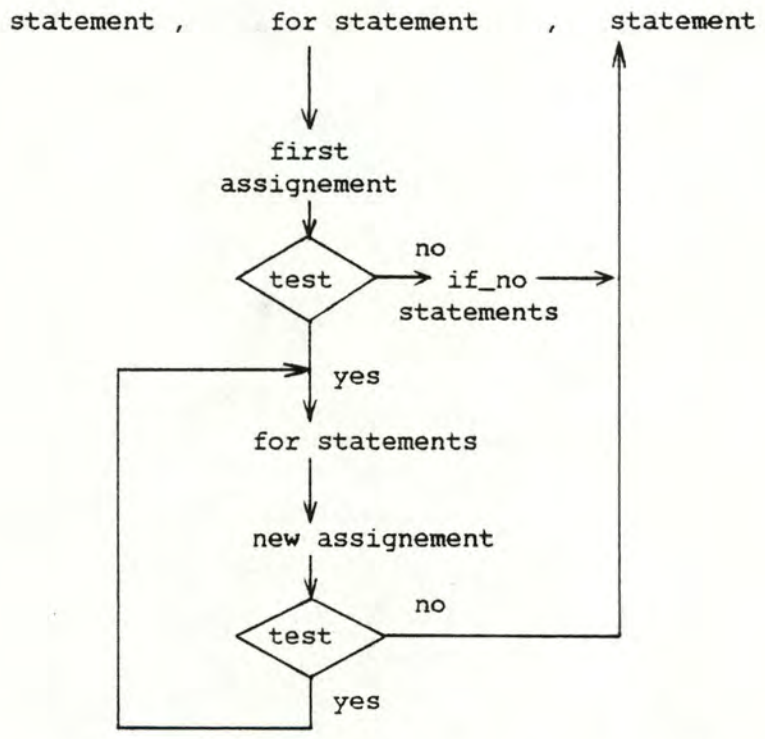
```
<for_st> ::= for <variable> := <for_list> do <statements>
                    <if_no_st> endfor

<for_list> ::= <collection_expression>
             | <collection_expression> <order>

<order> ::= order <order_keys>

<order_keys> ::= <name> <ad> , <order_keys>
               | <name> <ad>

<ad> ::= ascending
       | descending
       | <empty>

<if_no_st> ::= if_no <variable> then <statements>
             | <empty>
```

2.6.1.2. Example

```
NBCLI = 0;
for CLI:=CLIENT(CC:COM( :DATE_COM=X))order NAME_CLIENT do
        PRINT!(NAME_CLIENT);
        NBCLI:=NBCLI + 1
        if_no CLI then PRINT!(MESS_ERR)
endfor
```

2.6.1.3. Semantics

Like in the effective ADL, the for statement is an enumerative loop. In the predicative ADL, however, two sets of statements can be performed. The main difference with the effective ADL is that, when the test performed concludes that there is no element satisfying the selection condition of the for statement, the if_no statements are performed. This can be described as follows:



The set of values or references that can be assigned to the control variable are the elements selected by the for list. This set of possible elements is always ordered. The order of the sequence may be implicit (it is defined through the schema description) or it may be defined explicitly by means of the order clause. In this case, the sequence of access corresponds to a sorted order defined by one or more data items of the record type. This order clause may only be used if the collection expression denotes a DB object set (see section 1.3.1.2.2.).

2.6.2. Delete statement

2.6.2.1. Syntax

```
<del> ::= delete <name>
        |delete <name> <relation_operator> <unsigned_integer> <name>)
```

2.6.2.2. Example

```
delete CLI(CC:0 COM)
```

2.6.2.3. Semantics

The second way of deleting a record is new. The specified database record is deleted only if it satisfies the deleting condition, that is if it is linked to no record of the defined sequence via the access path type specified in the relation operator.

CHAPTER 4: THE WORKBENCH DATABASE

INTRODUCTION

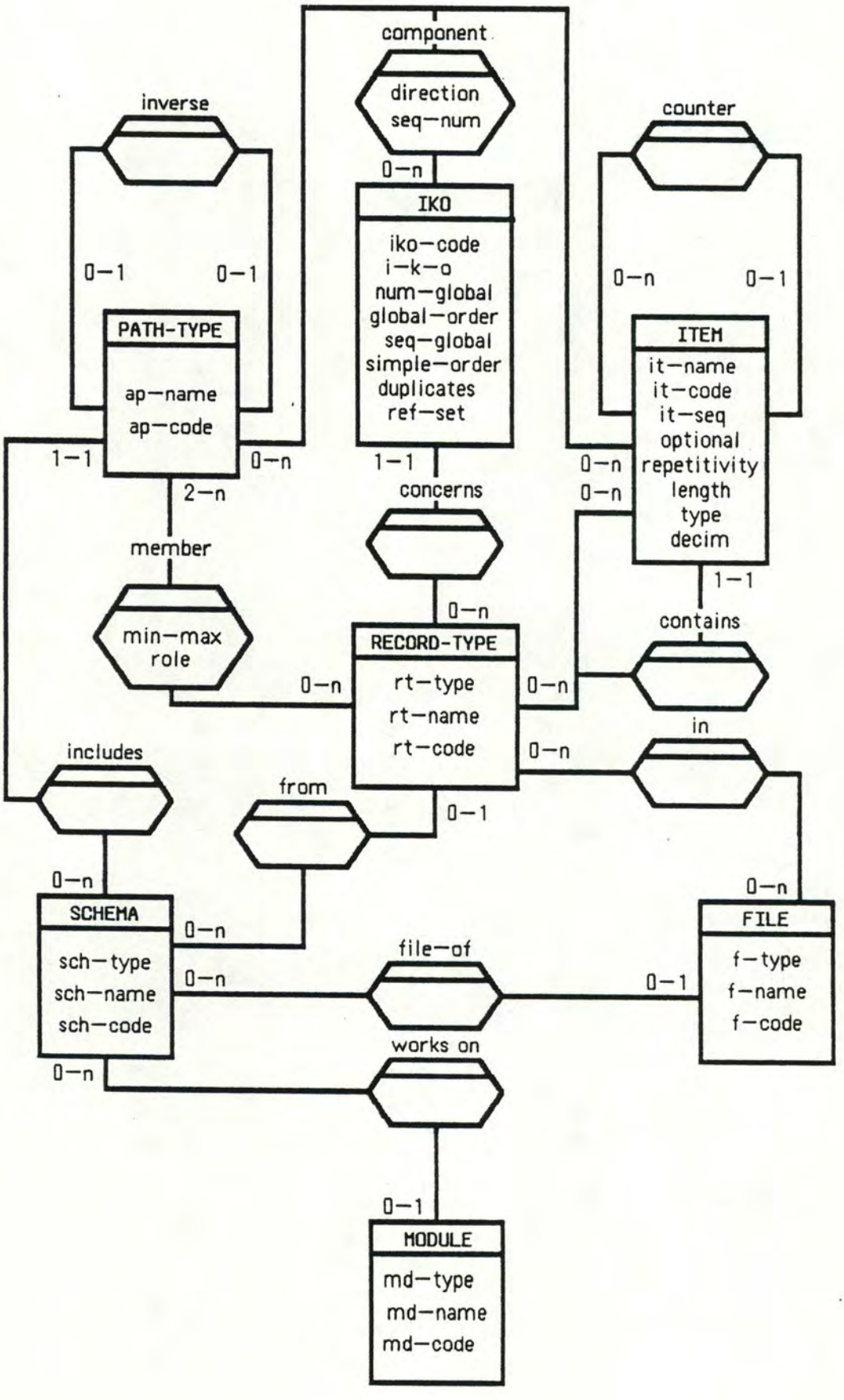
The developed workbench suggests an overall framework for database design at the logical layer independent of a particular target DBMS while progressing in the database design.

The database for the logical database design system has been designed to allow the representation of database schemata, expressed in the GAM model (see chapter 2). As the description of database schemata may be expressed in the GAM model too, the workbench database can be used at two different levels of representation.

In this chapter, we shall present :

- the conceptual schema of the database supporting the database design workbench; it is described in the E/R model
- the binary access schema that has been derived from the conceptual schema.

1. CONCEPTUAL SCHEMA OF THE WORKBENCH DATABASE



The conceptual schema consists of two parts :

- the first part concerns database schemata. It is composed of the entity SCHEMA, the GAM entities FILE, RECORD TYPE, PATH TYPE and ITEM. Finally, as we shall explain later, the entity IKO represents the GAM concepts of identifier, access key and order.

- the second part allows the representation of the programs which work on the database schemata. Presently, this part is reduced to the single entity MODULE.

In the following description of the conceptual schema, we often refer to chapter 2 for the definition of the GAM entities. Moreover, a detailed description of this conceptual schema in the META-language or ISLDM (Information System Language Definition Manager) is given in the appendix. For a good introduction to this Life cycle Support Systems (LSS) generator, please read chapter 6 of the thesis by Moentack & Delcourt ([DELC.84]).

1.1. Entities

1.1.1. SCHEMA

It represents a database schema expressed in the Generalized Access Model. A schema is a logical representation of data structures.

Attributes

- SCH-TYPE : this attribute indicates the type of the schema; a schema may be a possible access schema (PAS), a necessary access schema (NAS) or an effective access schema (EAS).

The definition of these three types of schemata is determined by the design process which is presented in [HAIN.83b], [HAIN.84b] and in [DELC.84].

A possible access schema is the systematic translation of the data structures expressed in the E/R model. Expressed with the GAM, the schema allows to perform all possible ways of access to the data structure, thus offering a convenient representation of the data for the logical design.

A necessary access schema is derived from the PAS by retaining only those data structures and means of access that are used by the effective algorithms.

Finally, an effective access schema is obtained by transforming the NAS in such a way that it is compatible with the target DBMS.

- SCH-NAME : the string of characters contains the name of the schema.

- SCH-CODE : the attribute gives the code which contains the schema within the database; it is used by the access functions of the GAM.

1.1.2 MODULE

It represents an algorithm written in the ADL language. The program works on the data structure defined by the schema associated to it.

Attributes

- MD-TYPE : this attribute indicates the type of the module; the possible values are effective module (EM) and predicative module (PM).

  Predicative modules specify the manipulations of the database records by giving the properties of the selected collections of data. They don't define any particular sequence of accesses to get to the data. The algorithms work on the data structures of the PAS.

  Derived from a predicative module, an effective module exactly specifies the way of acceding to the data. It describes the sequence of accesses necessary to reach the collection of database records defined by the expression in the corresponding effective algorithm.

- MD-NAME : the string of characters contains the name of the module.

- MD-CODE : this attribute contains the code which identifies the module within the database; it is used in the GAM access functions.

1.1.3. FILE

It represents a collection of database objects as it denotes an object FILE of the GAM (see section 1.4. of chapter 2).

Attributes

- F-TYPE : the attribute allows one to define the type of the database file.

- F-NAME : the string of characters denotes the name of the file.

- F-CODE : this attribute indicates the code of the file; it identifies the file within the database and is used in the GAM functions.

1.1.4. RECORD-TYPE

It represents a class of database records as it corresponds to the GAM object RECORD TYPE (see section 1.1. of chapter 2).

Attributes

- RT-TYPE : the attribute allows one to indicate the type of the class of records.

- RT-NAME : the string of characters denotes the name of the record type.

- RT-CODE : this attribute contains the code which identifies the record type within the database schema; it serves in the GAM access and manipulation functions.

1.1.5. PATH-TYPE

It represents an access path type of the database schema (see section 1.3. of chapter 2).

Attributes

- AP-NAME : the string of characters contains the name of the path type.

- AP-CODE : this attribute indicates the code of the access path type; it identifies the path type within the schema and is used in the access functions of the GAM.

1.1.6. ITEM

It represents a GAM object DATA ITEM (see section 1.2. of chapter 2).

Attributes

- IT-NAME : the string of characters denotes the name of the data item.

- IT-CODE : the attribute indicates the code of the data item which identifies it within the record type it is contained in; this code is used in the GAM functions.

- IT-SEQ : this attribute indicates the ordinal number of the data item on the given decomposition level.

- OPTIONAL : this attribute indicates if an instance of the associated record type or decomposable data item must (value = NO) or must not (value = YES) contain a value of this data item.

- REPETITIVITY : the attribute specifies if the data item is repetitive or not. A data item is repetitive if more than one

value of it can be associated to an instance of the record type or decomposable data item it is contained in. The values of the repetitivity property are :

0 if the data item is not repetitive
n if the repetitivity of the item is limited; this repetitivity is fixed if the item has no counter, variable otherwise.
999 if the repetitivity is unlimited.

- LENGTH : the attribute defines the length of the maximum size of the data item values.

- TYPE : by this attribute one defines the type of the data item; its value is either NUM (numeric), CHAR (character) or DEC (decomposable).

- DECIM : the attribute indicates the number of decimals for a data item of type numeric.

1.1.7. IKO

It represents the generalization of three concepts : Identifier, access Key, Order. We presented these concepts in the description of the GAM. Identifier is an integrity constraint (see section 3.2. of chapter 2), whereas access key and order are objects of the GAM (see section 1.6. and 1.7. of chapter 2).

An instance of an object type IKO may represent one, two or all three concepts at the same time.

Attributes

- IKO-CODE : the attribute indicates the code identifying the iko within the record type.

- I-K-O : the value of the attribute indicates if the iko denotes an identifier ("I") and/or an access key ("K") and/or an order ("O") for the record type that it concerns. In concrete terms, the value is a string of one to three letters taken from the alphabet {"I","K","O"}. The letters appear in that order, but one or the other may be skipped, meaning that the iko does not represent the concept the letter stands for. Thus the possible values are :
  IKO, IK, IO, I, KO, K, O.

- NUM-GLOBAL : The concept of global is created to extend the domain of validity of an iko to many record types of the database. Indeed, there may be record types which contain the same kind of information as they belong to a same class of objects at a higher level of abstraction. Hence, the data items of different record types which represent common information may constitute a global identifier, access key or order. In other

words, ikos that are common among several record types are said to belong to the same global.

A global is identified by an integer number. The attribute NUM-GLOBAL contains this number of the global. If however, the iko does not belong to any global, the value of NUM-GLOBAL is zero.

- GLOBAL-ORDER : this attribute defines the type of order of the global. Its values are :
  RANDOM, FIFO, LIFO, PRIOR, NEXT, SORTED.

- SEQ-GLOBAL : it indicates the ordering of the record types within a global order.

- SIMPLE-ORDER : it indicates the type of order within one record type. This property has the same set of possible values as the attribute GLOBAL-ORDER.

- DUPLICATES : the attribute points out if the iko representing a sorted order allows duplicates. If so, the attribute also gives the order of storage of the duplicates :
  NO : no duplicates
  RANDOM: duplicates stored in random order
  FIFO : duplicates stored in first-in-first-out order
  LIFO : duplicates stored in last-in-first-out order
  PRIOR : duplicates stored in a programmed order
  NEXT : duplicates stored in a programmed order

- REF-SET : it indicates which one of the three types of reference sets applies to the iko
  DB = the whole database
  EF = each file taken separately
  AP = the access path associated to the iko.

1.2. Relations

1.2.1. FILE-OF

This one-to-many relation from the entity SCHEMA to the entity FILE is used to express the fact that a particular file belongs to one schema. The relationship is optional for both entities.

1.2.2. FROM

This one-to-many relation from the entity SCHEMA to the entity RECORD-TYPE is used to link the record types to the schema they belong to. The relationship is optional for both entities.

1.2.3. IN

This many-to-many relation from the entity RECORD-TYPE to the entity FILE expresses the fact that a file is a collection of record types which, in turn, may be collected in more than one file.

1.2.4. CONTAINS

This relation links a record type or a decomposable data item to the data items belonging to it. It is a one-to-many relationship from the containing entity (RECORD-TYPE or ITEM) to the contained entity (ITEM). The relationship is mandatory for the data item; it either belongs to a record type or to a decomposable data item.

1.2.5. COUNTER

This one-to-many relation defined recursively on the entity ITEM is used to express the fact that a data item is the repetitivity counter of another data item which has to be repetitive. The relationship is optional.

1.2.6. CONCERNS

This relation is defined between the entity IKO and the entity RECORD-TYPE to relate a particular IKO to the record type concerned by this identifier, access key or order. The relationship is one-to-many from the record type to the IKO and it is mandatory for the IKO.

1.2.7. COMPONENT

This binary relation is defined from the entities ITEM and PATH-TYPE to the entity IKO. The many-to-many relationship expresses the fact that a data item is a component (it may be the only component) of an IKO and that an access path type belongs to the reference set of an IKO. (For example, a data item of a certain record type may be defined to be an identifier and an access key within the access path type leading to this same record type.) This relationship is optional for all three entities implied and it has two attributes:

- DIRECTION : for an IKO denoting an order, it indicates the nature of the order on the data items.
  ASC : ascending order
  DES : descending order

- SEQ-NUM : the attribute gives the sequence number of the data item among all the components of the IKO.

1.2.8. MEMBER

This many-to-many relation is defined from the entity PATH-TYPE to the entity RECORD-TYPE in order to link an access path type to its origin and target record types. A record type being a member of a path type if it is the origin or target of the path type, each path type has to be related to at least two record types. Hence, the relationship is double mandatory for the entity PATH-TYPE.

The relation has two attributes :

- ROLE : the attribute indicates if a record type is ORIGIN or TARGET of the related access path type.

- MIN-MAX : this attribute specifies the connectivity of the access path type. It gives the minimum and maximum number of times an instance of the considered record type is member in an access path of the same type. The possible values are :
  NM : none or more
  OM : one or more
  OO : one and only one
  NO : none or one.

1.2.9. INVERSE

This one-to-one relation is defined recursively on the entity PATH-TYPE to represent the fact that one access path type is the inverse of another one.

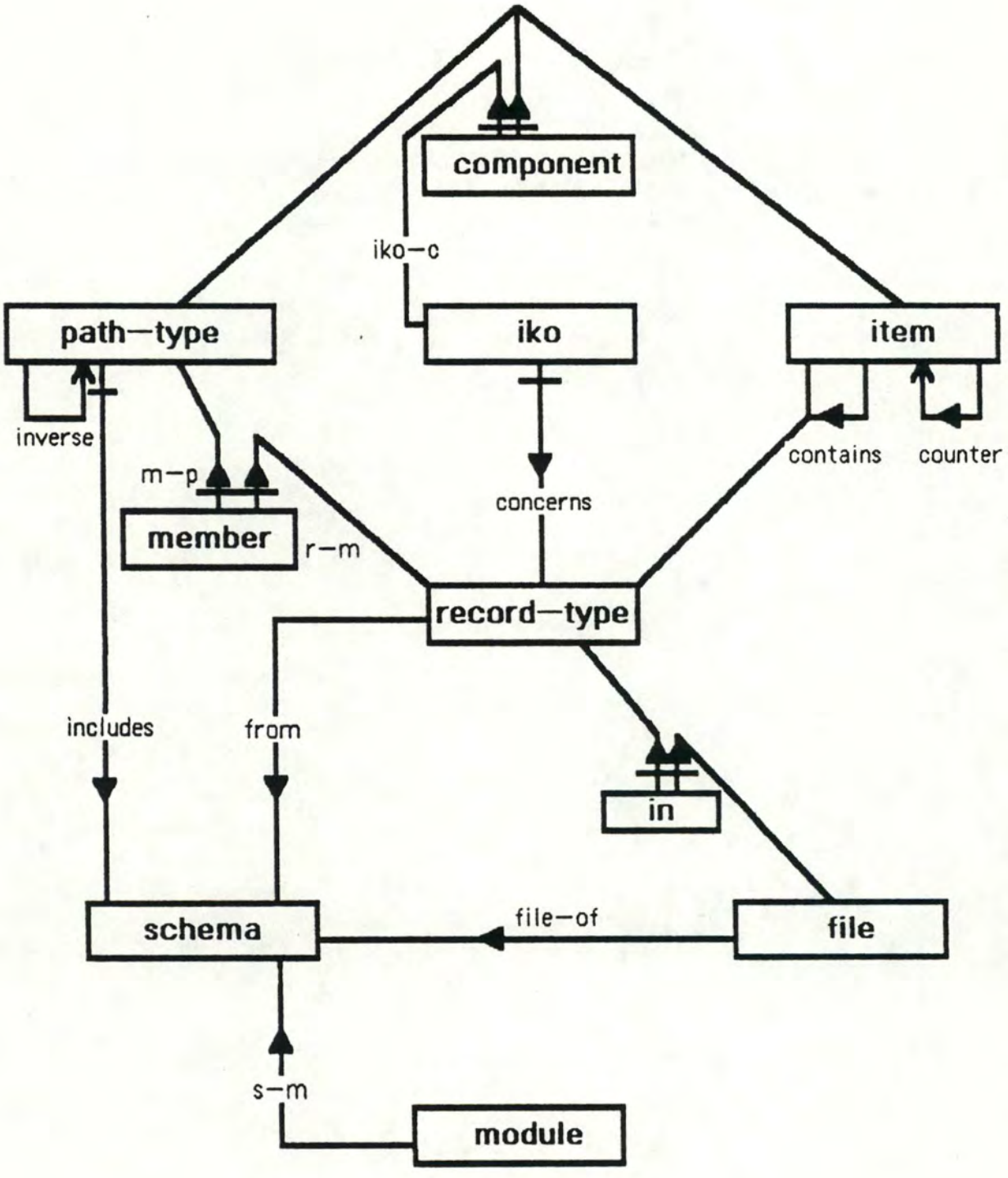
1.2.10. INCLUDES

This one-to-many relation from the entity SCHEMA to the entity PATH-TYPE is used to link the access path types to the database schema they belong to. The relationship is mandatory for the entity PATH-TYPE.

1.2.11. WORKS-ON

This one-to-many relation from the entity SCHEMA to the entity MODULE is used to associate the programs to the schema they work on. The relationship is optional for both entities implied.

2. BINARY ACCESS SCHEMA OF THE WORKBENCH DATABASE



The binary access schema is derived by a more or less systematic and direct translation of the data structure from the E/R model representation into the GAM formalism. This GAM representation has the advantage of expressing the data structure in an adequate way for the logical design of the various toolkits of the workbench.

More precisely, the transformation consists in the association of :

- a record type to each entity type
- a data item to each attribute of an entity type
- a path type with inverse to each type of binary relation with no attributes
- a record type, data items and two access paths (with inverse) identifying this record type to the binary relation types with attributes (see the MEMBER and COMPONENT relation).
  Being a binary relation with on one side two different entity types, the COMPONENT relation is transformed into a record type (called COMPONENT) and two access path types (IKO-C AND C-I-P) of which one has to allow two record types as origin, too. This means that a couple - data item, path type - combined with an IKO identifies one component.

All these transformations correspond in fact to the systematic translation rules of some data structure from the E/R model into the PAS expressed with the GAM. But the derivation of the binary access schema also applies some schema transformations to limit its generality :

- associate a record type and two access path types (with inverse) identifying this record type to a many-to-many path type (see the record type IN). This transformation allows us to avoid a many-to-many access path type, a type of access rarely provided by a real DBMS. Hence the transformation adds to the portability of the database schema.

- add a data item to the record type PATH-TYPE to indicate if a particular path type is a defined access path type or the inverse of some defined path type. Because of the recursivity of the INVERSE relationship, it is necessary to offer a way of knowing what kind of path type one is looking for when asking the inverse of some path type. The possible values of this indicator is 0 and 1. If this data item is 1 for an access path type, it means that this path is origin of an access path type INVERSE. For its inverse access path type, this data item is 0 and it indicates that the access path type is origin of a path type INVERSE-I.

The application of these general transformation rules produces a binary access schema which contains the following record types :

SCHEMA, FILE, RECORD-TYPE, ITEM, IKO, PATH-TYPE and MODULE which exactly represent the types of objects denoted by the entity types they have been derived from. Thus the data items of these record types correspond to the attributes defined for the entity types. The only exception is the one data item of the record type PATH-TYPE which we mentioned earlier.

IN, COMPONENT and MEMBER which are created to represent a many-to-many relation or a binary relation with attributes. The record type IN has no data items whereas the data items of COMPONENT and MEMBER denote the attributes of the relationships they are standing for.

In addition, the access schema includes all the access path types derived from the relations. All these access path types have an inverse. There is one one-to-one access path type (INVERSE); all the other path names marked on the schema denote one-to-many access path types. It follows that their inverse path types denote many-to-one path types. The name of an inverse access path type is formed by adding the string "-I" at the end of the name of the path type it is the inverse of.

A more detailed presentation of the access schema of the workbench which includes a complete description of the structures of the record types as well as a list of the path types with their properties is given in the appendix.

CHAPTER 5: GENERAL TOOLS USED TO IMPLEMENT THE ADL PARSER.

1. INTRODUCTION.

This chapter presents the choice that was made to design the first part of the ADL analyser; that is the parser. The ADL parser receives a source text written in ADL and produces a parse tree. The second part, the semantic analyser will be presented in the next chapter.

The activities of the first part of the analyser may be decomposed in a lexical analysis, a syntactical analysis and the construction of the parse tree.

The UNIX system provides two important tools that prevent us from writing the lexical and the syntactical analyser from scratch.

Indeed, the development of new programs on the UNIX system is largely facilitated by tools for language design and implementation. These are frequently programs generators, compiling into C, which provide advanced algorithms in a convenient form. Two of the most important tools are LEX, a generator of regular expressions recognisers using deterministic finite automata, and YACC a generator of LALR(1) parsers (Look Ahead Left Right). Section 2 will present what is a lexical analyser and LEX; while section 3 will present what is a syntactical analyser and YACC.

Section 4 presents the tree structure that was used for the parse tree. It will also present the parse tree construction functions as they were defined in Namur by Michel Buyse. We call all these tools the SYNTACTICAL ENVIRONMENT.

Another important tool developped also by Michel Buyse is a general tool called the transformator, which receives the description of a certain formalism in a special language LDF (formalism definition language). It produces all the informations necessary so that the syntactical environment can be used to generate an analyser for the texts expressed in that formalism. Section 5 will present the formalism definition language and section 6 the general definition of the transformator [BUYS.85a] and [BUYS.85b].

All these tools were used to implement the ADL analyser that produces the parse tree.

2. LEXICAL ANALYSIS AND LEX.

2.1. The lexical analysis.

The lexical analysis is the interface between the source program and the syntactical analyser. It reads the source program one character at a time carving that program into a sequence of atomic units called tokens. A token consists of a sequence of characters and can be considered as a single logical entity.

In the following example, the ADL statement can be decomposed into 20 tokens.

```
for A:=1..10 do
   B:=A*A;
   PRINT!(A,B)
 endfor
```

These tokens are:

for	B	(
A	:=	A
:=	A	,
1	*	B
..	A	)
10	;	endfor
do	PRINT!	

2.2 LEX.

LEX is a tool available on the UNIX system. It is a program generator designed for lexical processing of character input streams. It accepts a high-level problem oriented specification for character string matching, and produces a program in a general purpose language (RATFOR or C) which recognizes regular expressions. C is the language that was chosen.

Thus, LEX source is a table of regular expressions and corresponding program fragments. The table is translated into a program that reads an input stream, copying it to an output stream and partitionning the input stream into strings matching the given expression.

A LEX specification is divided in 3 parts:

```
 definition of the lexical sets
%%
 rules
%%
 functions defined by the user
```

The definition and function parts are not important and are often omitted.

The rules part contains a set of rules. They represent the user's control decisions. The rules are in a table, in which the left part contains a target and the right part contains actions (program fragments) to be executed when the target is encountered.

Targets.

There are two kinds of targets: literals and regular expressions.

- a literal is a set of predefined characters. For example IF is validated by the sequence of characters "IF" and only by that one;
- a regular expression allows the definition of a target by means of the set of characters accepted. So, for example, [a_zA_Z][a_zA_Z0_9]* is validated by the sets of characters "VAR", "var", Var9"...

Actions.

The actions to be executed must be written in C. If we want to use LEX with the other tool YACC, each action must end with "return(name)", where name is a shared identifier with YACC. Name is also a means to identify the token just read.

For example, it is a means to know that the reserved word "IF" has

just been read; the shared identifier can be, for example, res_word_if.

So, in the case of literals, the lexical analyser will only send to the syntactical analyser generated by YACC, the type of the token that was just read. The syntactical analyser needs to know that the reserved word "IF" has been read and does not need the string IF itself.

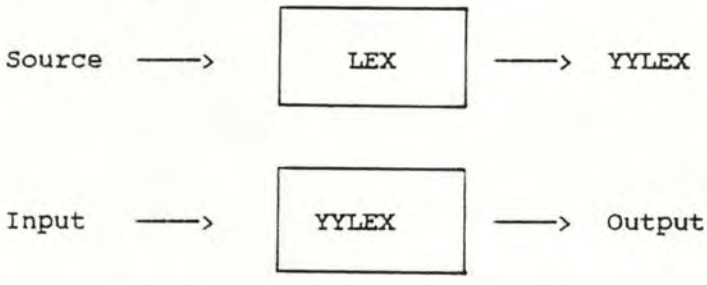
However, when the target is a regular expression, the syntactical analyser needs to know both the type of the token and its value. Indeed, knowing that an identifier has been read is not enough; it is also important to know the name of that identifier.

The order of the rules is significant. If two targets are validated by the same set of characters in the input, the choice between the targets is done as follow:

- the longest validation is preferred (in other words, the target that accepts the greatest number of characters);
- among the rules that validate the same number of characters, the rule in first place is preferred.

Conclusion on LEX.

LEX can be described as follows:



YYLEX is the generated program, that will recognize targets in a stream (input) and will perform the specified action for each target detected.

3. SYNTACTICAL ANALYSIS AND YACC.

3.1. The syntactical analysis.

The syntactical analysis has two functions. First, it checks that the tokens that appear in its input (that is the output of the lexical analyser) occur in patterns that match the specification of the source language. It also imposes on the tokens a tree structure, that means that the syntactical analyser calls the parse tree construction functions, so the intern representation of the text is created.

Here is a simple example of the first task of the syntactical analyser; the second will be explained later.

If an ADL program contains the expression

```
while A do do
```

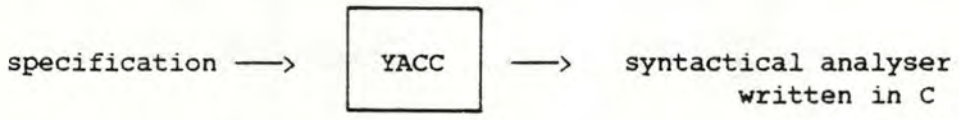
then, after the lexical analysis, that expression appears to the syntactical analyser as the tokens sequence

```
while name do do .
```

On seeing the second do, the syntactical analyser should detect an error, because the presence of two do adjacent violates the formation rule of a while statement.

3.2. YACC.

YACC is a program generator available on the UNIX system. It provides a general tool for imposing structure on the input of a computer program. The YACC user prepares a specification of the input process. From that, YACC generates a function to control the input process.



A YACC specification is divided in three parts:

```
 definitions
%%
 rules
%%
 functions
```

The functions part is not important and is often empty; it won't be discussed here.

First part: the definition part of a YACC specification.

This area serves to define the names used as token names in the rules part. These names are the ones that we find in the LEX specification as arguments of the return functions associated with the description of the lexical entities.

Example: %token tif tthen telse ...

In this part, we need to define the starting point of the analysis; that is the most global structure of a formalism.

Second part: the rules part.

This part is composed of a set of rules. Each rule consists of three parts: the entity defined, its definition and the actions to be executed when that entity is encountered.

- the entity defined corresponds to the left member of a concret syntax rule expressed in BNF (Backus-Nauer Form).
- the right member of the rule is formed by the description of the entity. In this description, the reserved words and separators have been replaced with a token name shared with the lexical analyser.

Example.

Concrete syntax rule:

<if> ::= if <cond> then <stat> else <stat>

```
LEX:    if      return(tif);
        then    return(tthen);
        else    return(telse);
```

YACC: if: tif cond tthen stat tesle stat

- the action to be executed is the construction of the corresponding tree. In those actions, it is possible to use YACC's meta-variables. These meta-variables have the form "$x", where "x" is either a number or the symbol "$".
  "$$" is the meta-variable which receives a value in the current rule. "$i" is the way to obtain the value affected to the term in position i in the right member of the rule. This process is the derivation process.

  Example.

  In the above described rule, $2 allows us to obtain the value affected when the rule cond is recognized; $4: the same applies in the case of the first statement rule and $6 for the second statement rule. $$ allows to affect a value to the current rule, in this case the 'if' rule. These values are references to the abstract tree where these different terms of the rule are stored.

  The entire rule is thus:

```
if: tif cond tthen stat telse stat
   { $$ = node_constr(if_code,$2,$4,$6);};
```

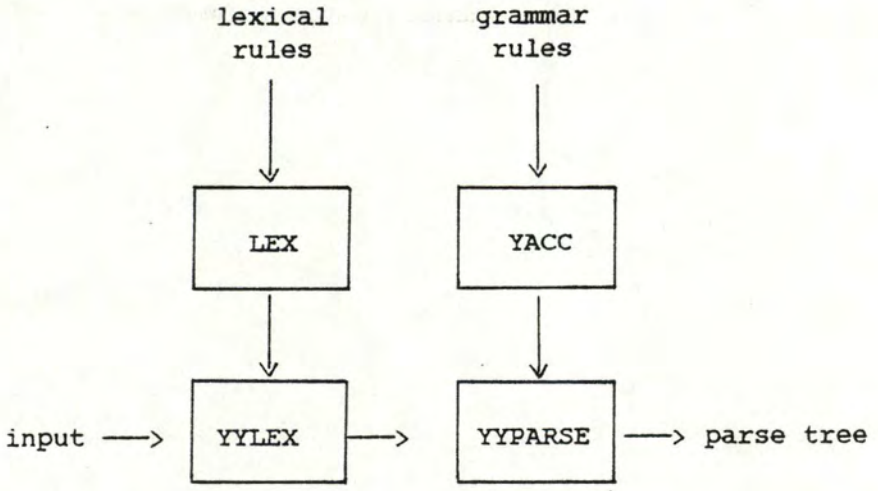
  All the actions performed, when a syntactical entity is recognized, are the parse tree construction functions explained in the section 4.

It is necessary to define a particular rule, that will serve as starting point for the analyser. It is from that starting point that YACC is going to try to validate the tokens sequence. The starting point represents the most global structure in the defined formalism.

3.3. Cooperation of LEX & YACC.

So, as described in the preceding section, the input data text read is divided into tokens by the lexical analyser generated by LEX. They are communicated to the parser which organizes them into larger structures.

The cooperation between LEX and YACC can be described as follows:



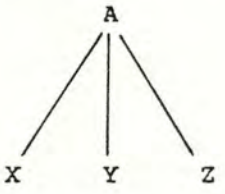
4. THE PARSE TREE CONSTRUCTION FUNCTIONS.

4.1. Tree definitions.

As the parser produces a tree, it seems important to define what is a tree. An important work was done on that topic by Moentack & Delcourt [DELC.84]. Here let's brievly recall the major characteristics of the tree structure.

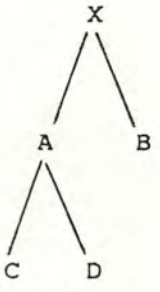
A parse tree is a tree where each node is labelled with some non terminal A, and the children of that node are labelled from left to right with the symbols of the right side of the production by which A was replaced in the derivation process at the syntactical analysis.

For example, if A->XYZ is a production used at some step of a derivation, the parse tree for that derivation will have the subtree

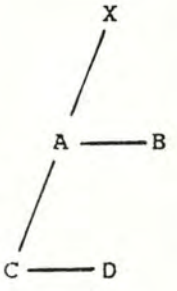


The leaves of the parse tree are labelled with non terminals or terminals and are read from left to right. They constitute a sequential form called the yield of the tree.

The trees implemented are binary trees. Ordinary trees can be transformed in binary trees by linking together the sons of each family and removing vertical links except from the father to his first son. This is explained in the following example.

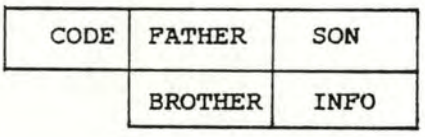


Ordinary tree.



Binary tree.

In the parse trees, all nodes have the same structure. A node can contain 5 different informations. The nodes have the following structure.



The meaning of each field is the following:

- CODE : field dedicated to contain the code of the node;
- FATHER : field that contains the reference of the node that is the father of the described node, NIL if it has no father so far;
- SON : field that contains the reference of the node that is the son of the described node, NIL if it has no son so far;
- BROTHER :field that contains the reference of the node that is the brother of the described node, NIL if it has no brother so far;
- INFO : field dedicated to the user.

More information on trees can be found in [DELC.84].

4.2. Parse tree construction functions.

In this section, we will describe the functions that are used in the YACC rules to build the parse tree.

These functions are:

- NUN: construction of an unary node;
- NBIN: construction of a binary node;
- NTER: construction of a ternary node;
- NQUAT: construction of a quaternary node;
- NZER: construction of a zeroary node;
- CLIEN: construction of a link between two list members;
- HLISTE: construction of a list head node;
- CONSTRCOMM: construction of a comment node;
- CONSTRGEN: construction of a generic node;
- CONSTRIDENT: construction of an identifier node;
- CMETA; construction of a meta node.

All these functions were defined by Michel Buyse, they will be described in the sequel and more information can be found in [BUYS.84].

FUNCTION 1: NUN creation of an unary node.

INPUT.

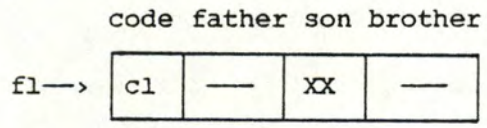
- t: code of the node to create;
- f1: reference to a node in the tree where the information of the son of the node to create are stored.

OUTPUT.

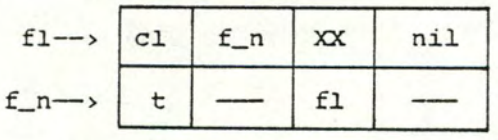
NUN returns the reference of the node that was just created. That means that:
- a node, whose code is t and whose son is f1, is created in the tree;
- a father link is created between f1 and the new node.

Example.

Before the call.



After the call f_n = nun



FUNCTION 2: NBIN creation of a binary node.

INPUT.

- t: code of the node to create;
- f1: reference to a node in the tree where the information of the first son of the node to create are stored;
- f2: reference to a node in the tree where the information of the second son of the node to create are stored;

OUTPUT.

NBIN returns the reference of the node just created. That means that:
- a node, whose code is t and whose son is f1, is created in the tree;
- a father link is created between f1 and the new node and between f2 and the new node;
- a brother link is created between f1 and f2.

Example.

Before the call.

	code	father	son	brother
f1-->	c1	---	X1	---
f2-->	c2	---	X2	---

After the call f_b = nbin

	code	father	son	brother
f1-->	c1	f_b	X1	f2
f2-->	c2	f_b	X2	nil
f_b-->	t	---	f1	---

FUNCTION 3: NTER creation of a ternary node.

INPUT.

- t: code of the node to create;
- f1: reference to a node in the tree where the information of the first son of the node to create are stored;
- f2: reference to a node in the tree where the information of the second son of the node to create are stored;
- f3: reference to a node in the tree where the information of the third son of the node to create are stored.

OUTPUT.

NTER returns the reference of the node just created. That means that:
- a node, whose code is t and whose son is f1, is created in the tree;
- a father link is created between f1 and the new node and between f2 and the new node and between f3 and the new node;
- a brother link is created between f1 and f2 and between f2 and f3.

Example.

Before the call.

	code	father	son	brother
f1—>	c1	——	X1	——
f2—>	c2	——	X2	——
f3—>	c3	——	X3	——

After the call f_t = nter.

	code	father	son	brother
f1—>	c1	f_t	X1	f2
f2—>	c2	f_t	X2	f3
f3—>	c3	f_t	X3	nil
f_t—>	t	——	f1	——

FUNCTION 4: NQUAT creation of a quaternary node.

INPUT.

- t: code of the node to create;
- f1: reference to a node in the tree where the information of the first son of the node to create are stored;
- f2: reference to a node in the tree where the information of the second son of the node to create are stored;
- f3: reference to a node in the tree where the information of the third son of the node to create are stored.
- f4: reference to a node in the tree where the information of the fourth son of the node to create are stored.

OUTPUT.

NQUAT returns the reference of the node just created. That means that:

- a node, whose code is t and whose son is f1, is created in the tree;
- a father link is created between f1 and the new node and between f2 and the new node and between f3 and the new node and between f4 and the new node;
- a brother link is created between f1 and f2 and between f2 and f3 and between f3 and f4.

Example.

Before the call.

	code	father	son	brother
f1-->	c1	——	X1	——
f2-->	c2	——	X2	——
f3-->	c3	——	X3	——
f4-->	c4	——	X4	——

After the call f_q = nquat.

	code	father	son	brother
f1—>	c1	f_q	X1	f2
f2—>	c2	f_q	X2	f3
f3—>	c3	f_q	X3	f4
f4—>	c4	f_q	X4	nil
f_q—>	t	——	f1	——

FUNCTION 5: NZER construction of a zeroary node.

INPUT.

- t: code of the node to create.

OUTPUT.:

NZER returns the reference of the node just created

Example.

Before the call.

——

After the call n_z = nzer.

	code	father	son	brother
n-z—>	t	——	nil	——

FUNCTION 6: CLIEN creation of the brother link between two list members.

INPUT.

- f1: reference of the preceeding list element;
- f2: reference of the current list element.

OUTPUT.

CLIEN returns the reference of the current list element. That means that a brother link is created between the preceeding element and the current element.

Example.

Before the call.

	code	father	son	brother
f2 —>	cj	——	xj	——
f1 —>	ci	——	xi	fi-1
fi-1—>	ci-1	——	xi-1	fi-2
....				....
fi-n—>	ci-n	——	xi-n	nil

After the call clink = clien.

	code	father	son	brother
clink—>	cj	——	xj	f1
f1 —>	ci	——	xi	fi-1
fi-1—>	ci-1	——	xi-1	fi-2
....				....
fi-n—>	ci-n	——	xi-n	nil

Remark: in such a list the first element is actually the last element of the list, and so on.

FUNCTION 7: HLISTE creation of the head of the list.

INPUT.

- t: the code of the node to create;
- fl: reference of the last element of the list analysed.

OUTPUT.

HLISTE returns the reference of the node just created. That means that:

- a father link is created between all the elements of the list and the new node;
- the order of the elements of the list is restored.

Example.

Before the call.

	code	father	son	brother
fl —>	ci	——	xi	fi-1
fi-1—>	ci-1	——	xi-1	fi-2
....				....
fi-n—>	ci-n	——	xi-n	nil

After the call h_l = hliste.

	code	father	son	brother
h_l—>	t	——	fl	——
fi-n—>	ci-n	h_l	xi-n	fi-n+1
....				....
fl —>	ci	h_l	xi	nil

FUNCTION 8: CONSTRCOMM creation of a comment node.

INPUT.

- t: code of the node to create;
- txt: array that contains the comment to memorize;
- len: lenght of that comment.

OUTPUT.

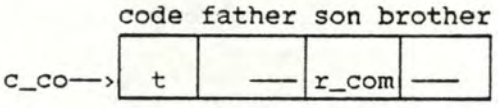
CONSTRCOMM returns the reference of the node just created. The text of the comment can be accessed via the reference contained in the son of the node.

Example.

Before the call.

—

After the call c_co = constrcomm.



FUNCTION 9: CONSTRGEN creation of a generic node.

INPUT.

- t: code of the node to create;
- txt: array that contains the generic to memorize;
- len: lenght of that generic.

OUTPUT.

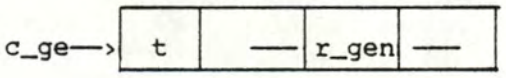
CONSTRGEN returns the reference of the node just created. The text of the generic can be accessed via the reference contained in the son of the node.

Example.

Before the call.

—

After the call c_ge = constrgen



FUNCTION 10: CONSTRIDENT creation of a identifier node.

INPUT.

- t: code of the node to create;
- txt: array that contains the identifier to memorize;
- len: lenght of that identifier.

OUTPUT.

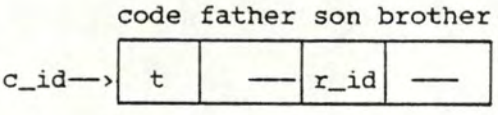
CONSTRIDENT returns the reference of the node just created. The text of the identifier can be accessed via the reference contained in the son of the node.

Example.

Before the call.

—

After the call c_id = constrident



FUNCTION 11: CMETA construction of a meta node.

INPUT.

- t: code of the meta node in the current formalism;
- cno: code of the node for which we want to create the meta instance.

OUTPUT.

CMETA returns the reference of the meta node just created. That means that:

- a node of code t and a node of code cno are created;
- a father-son link is created between them.

Example.

Before the call.

—

After the call cm = cmeta.

	code	father	son	brother
cm —>	t	—	p	—
p —>	cno	cm	nil	nil

p is the reference of the son of the node cm.

5. THE FORMALISM DEFINITION LANGUAGE (LDF).

5.1. Introduction.

LDF is a meta-formalism that is used to descript formalisms. A LDF program is the description of the syntax of a given formalism. The description is used to obtain an instantiation of the syntactical environment for that formalism. That is the lexical analyser, the syntactical analyser as well as the parse tree construction functions and some other tools that will be presented in this section (LDF) and the next section (the transformator).

The definition of a new formalism (that is the text introduced) should to be brief. This definition allows to generate the syntactical environment for the described formalism without any extra informations.

5.2. General presentation of the LDF language.

The LDF programs syntax is very closed to the BNF notation.
Non terminals of the language are placed between "<" and ">".

A LDF program consists of a heading followed by a semi-colon and a body ending with a point.

A) The heading.

The heading consists of a static part, pointing out that the text analysed is expressed in LDF (*debut ldf*), and a dynamic part indicating the name of the defined formalism.

Example.

```
(* debut ldf *)
definition de adl;
```

B) The body.

The body of a LDF program consists of a set of production rules separated by semi-colons.

A production rule consists of a left part containing one and only one non terminal (NT) and a right part consisting of zero, one, two, three or four NTs. There is an exception: alternative rules can contain a boundless number of NTs in the right part.

A NT can appear in one and only one left member of a rule. There are reserved NT the meaning of which will be explained later.

The order of the rules is not important.

A rule definition can be:

- a list rule;
- a quaternary rule;
- a ternary rule;

- a binary rule;
- a unary rule;
- a zeroary rule;
- a generic rule.

All these rules will be presented in the sequel; but LDF programs can include other things. Section 5.4 of this chapter will present some informations about the non terminals, the separators, the decompilation symbols and some reserved NTs.

5.3. Rules.

5.3.1. Definition of the list rule.

It is allowed to define two kinds of lists; a list that can be empty and a list that must at least have one element.

A) List that can be empty.

-> LDF formulation: L<NT> ::= <NT2>*sep1
Where L indicates that it is a list rule;
NT is the non terminal to define;
NT2 is the non terminal that is the element of a NT list;
* indicates that the list can be empty;
sep1 is the separator of the elements of the list.

Example.

L<statements> ::= <statement>*";" ;
defines
- (empty list)
- statement
- statement ; statement ; statement

B) List with at least one element.

-> LDF formulation: L<NT> ::= <NT2>+sep1 ;

Where L, NT, NT2, sep1 have the same meaning as in the preceding defined kind of list and where + indicates that the list must at least have one element.

5.3.2. Definition of a quaternary rule.

A quaternary rule is a rule whose right member contains four NTs.

-> LDF formulation:

```
<NT> ::= sep1 <NT1> sep2 <NT2> sep3 <NT3> sep4 <NT4> sep5 ;
```

Example.

```
<for_st> ::= "for" <variable> ":=" <collection_expression> "do"
                <statements> "if_no" <statements> "endfor" ;
```

5.3.3. Definition of a ternary rule.

A ternary rule is a rule whose right member contains three NTs.

-> LDF formulation:

```
<NT> ::= sep1 <NT1> sep2 <NT2> sep3 <NT3> sep4 ;
```

Example.

```
<if_then_else> ::= "if" <general_expression> "then" <statements>
                   "else" <statements> "endif" ;
```

5.3.4. Definition of a binary rule.

A binary rule is a rule whose right member contains two NTs.

-> LDF formulation:

```
<NT> ::= sep1 <NT1> sep2 <NT2> sep3 ;
```

Example.

```
<if_then> ::= "if" <general_expression> "then" <statements> "endif" ;
```

5.3.5. Definition of an unary rule.

There are three kinds of unary rules:
- alternative rules;
- simple rules;
- rules to which no tree constructors are associated with.

A) Alternative unary rule.

An alternative unary rule is a rule whose right member consists of only NTs separated by the symbol "|" meaning "or". An alternative rule serves to present the different productions of the left member NT. Alternative rules must be preceded by the string "ALT".

-> LDF formulation.

```
ALT<NT> ::= <NT1> | <NT2> | ... | <NTi> ;
```

Example.

```
ALT<statement> ::= <if_then_st> | <while_st> | ... | <call_st> ;
```

B) Simple unary rule.

A simple unary rule is a rule whose right part contains one NT.

-> LDF formulation.

```
<NT> ::= sep1 <NT1> sep2 ;
```

Example.

```
<statement_part> ::= "begin" <statements> "end" ;
```

C) Unary rule without tree constructor.

A rule without tree constructor (indicated by SC) is to be used, when we don't want to have in the tree a node of that type. This can be done so that, unnecessary informations won't be present in the tree.

-> LDF formulation:

```
SC<NT> ::= <NT1> | <NT2> | ... | <NTi> ;
```

Example.

```
SC<simple_data_type> ::= <integer> | <real> | <boolean> | <string> |
<numeric> ;
```

5.3.6. Definition of a zeroary rule.

A zeroary rule is used to allow the definition of reserved words of the language. There is no NT in the right member of those rules.

-> LDF formulation:

```
<NT> ::= sep ;
```

Example.

```
<integer> ::= "integer" ;
```

5.3.7. Definition of generic rules.

Generic of the language are elements that need to be represented as regular expressions.

-> LDF formulation:

```
GEN<NT> ::= "regular expression" ;
```

Where GEN indicates that it is the definition of a generic; the regular expression has the form of the regular expressions recognized by LEX. If '"' is present it needs to be represented twice: '""'.

Example.

```
GEN<unsigned_integer> ::= "[0_9]*" ;
```

5.4. Non terminals, separators, decompilation symbols and reserved words.

This section presents some informations about non terminals, separators, decompilation symbols and reserved words.

A) Non terminals.

All non terminals have to be recognized by the regular expression [a_zA_Z][a_zA_Z0_9_]*.

Examples: whilest, wh_st, st_NB9

B) Separators.

All separators are either empty or composed of a string of characters placed between quotation and whose lenght is not equal to zero.

C) Decompilation symbols.

The decompilation of a formalism is the process which reconstructs the ADL program text up from the parse tree stored in the system. It allows to visualize an algorithm which may have been transformed by tree manipulations. It may be visualize by two means. First, a possible option is to present that text clearly, with, for example, a statement beginning at each new line, with tabulations, and so on. The second way is to present the tree like an hierarchical decomposition. The two kinds of decompilation are given for the example in appendix.

To achieve the first kind of decompilation, are needs to include in the LDF formalism decompilation characters.

The characters $, #, @, ^ placed before (or after) a separator have the following meaning:

- $: a new line must begin before (or after) the impression of the separator;
- #: a new line must begin with a tabulation before (or after) the impression of the separator. The tabulation will only disappear after the decompilation of the NT following the symbol #;
- @: i lines must be skipped before (or after) the impression of the separator;
- ^: no blanks have to be inserted before (or after) the impression of the separator.

D) Reserved non terminals.

There are a small number of reserved non terminals that have a particular meaning.

i) the starting point.

The most important reserved non terminal is the starting point (point_d_entree).

<point_d_entree> ::= <program_heading> <program_body> "." ;

This non terminal represents the most global structure described by the grammar rules.

ii) annot.

That NT cannot be found in the left member of a rule (its definition is extern and it is not described in the current formalism). It can be found in several right members of some rules. When this NT is encountered, it means that the current language syntax allows the insertion, at that place, of a text fragment expressed in another formalism than the one currently described. But it was not used in the LDF definition of ADL.

iii) comment

That NT is provided to define comments in the current formalism. That NT can be found in several right members of the rules; and if it is present in at least one right member, we need to define that NT as a generic with a regular expression. But it was not used in the LDF definition of ADL.

Example: GEN<comment> ::= "(*[a_zA_Z][a_zA_Z0_9]+*)" ;

iv) ident.

For intern reasons, it is interesting to make a distinction among all generics, so that the identifier generic can be pointed out. Its definition must be present in the LDF program. The utility of that distinction is to make use of that generic in the recognition of generics of which we can find several identical instances in the text analysed. But it was not used in the LDF definition of ADL.

Example: GEN<ident> ::= "[a_zA_Z][a_zA_Z0_9]*" ;

5.5. Conclusion.

The formalism definition language described in the present section was used with the formalism ADL, so that it was possible to use the syntactical environment to generate an ADL parser. The way this has been achieved is described later on. The next section presents the transformator. The LDF definition of ADL can be found in the appendix.

6. THE TRANSFORMATOR.

6.1. Introduction.

The transformator is the tool that receives the description of a formalism F in the formalism definition language and from that description, generates all informations necessary to use the syntactical environment for texts expressed in the formalism F.

These informations are mainly the specification for LEX and YACC, the informations for the decompilation and the abstract syntax. This section will explain the first kind of informations.

6.2. Specifications for LEX & YACC.

A) Creation of the LEX specifications.

As seen in section 2.2., LEX analyses the specification and produces a lexical analyser for the formalism described.

Here are the different steps.

i) Lexical recognition of the meta nodes.

For each non terminal, if the string just read is a non_terminal_name, generate:

```
node name return(tok#);
```

Where

- node name is the name of the non terminal of the left part of the rule;
- # is a value from the range 1..p , where p is the number of rules and thus the number of meta nodes.

ii) Lexical recognition of the separators.

Here are the steps of that recognition:

- make a list of all distinct separators (words and reserved symbols) present in the LDF program;
- for each element of the list, generate:

```
separator return(tok#);
```

Where
- separator is the current element of the list of separators;
- tok# is an object to which YACC will give a value; # is a value in the range p+1..p+n (n is the number of distinct separators); two distinct separator receive a distinct value.

iii) Lexical recognition of the generics.

For each generic described in the formalism, generate:

```
regular expression   {return(tok#);};
```

Where
- regular expression is the lexical definition of the generic;
- tok# has the same meaning as tok# for the separators; # takes a value between p+n+1..p+n+m, where m is the number of generics of the formalism.

B) Creation of the YACC specifications.

The YACC specifications is a file that will be read by YACC so that a syntactical analyser will be generated for the formalism.
Here are the steps.

Step 1.

Declaration of the tokens; generate:

```
%token tok1 tok2 ... tokn+m+p
```

Step 2.

Declaration of the starting point of the analyser; generate:

```
%start XXXX
```
(XXXX is an intern reserved non terminal)

Step 3.

Each syntax rule receives an intern code depending on its class. The different codes given are the following:

- 1..a1: lists that can be empty;
- a1+1..a2: lists with at least one element;
- a2+1..a3: quaternary nodes;
- a3+1..a4: ternary nodes;
- a4+1..a5: binary nodes;
- a5+1..a6: unary nodes;
- a6+1..a7: zeroary nodes;
- a7+1..a8: generic nodes (except comments and identifiers);
- a8+1: annotations;
- a8+2: comments;
- a8+3: meta_nodes;
- a8+4: identifiers.

Step 4.

Definition of the production rules of YACC.

The syntax rules are to be transformed and actions are to be added. These actions are the ones specified in the tree construction functions. Each rule created includes the recognition clause of the rule at the meta level (cmeta rule).

1) Starting point of the analyser.

It points out that the tree construction functions can begin to be used.
The non terminal reserved has been recognized.
Example.

```
XXXX : ... { ...
         sommet($$);
         return(...);}
```

2) Quaternary nodes.

LDF syntax: <NT> ::= sep1 <NT1> sep2 <NT2> sep3 <NT3> sep4 <NT4> sep5;

Will produce:

```
NT : tok43 NT1 tok23 NT2 tok78 NT3 tok04 NT4 tok55
         {$$=nquat(code_NT,$2,$4,$6,$8);};
```

3) Ternary nodes.

LDF syntax: <NT> ::= sep1 <NT1> sep2 <NT2> sep3 <NT3> sep4;

Will produce:

```
NT : tok43 NT1 tok23 NT2 tok78 NT3 tok04
         {$$=nter(code_NT,$2,$4,$6);};
```

4) Binary nodes.

LDF syntax: <NT> ::= sep1 <NT1> sep2 <NT2> sep3

Will produce:

```
NT : tok43 NT1 tok23 NT2 tok78
         {$$=nbin(code_NT,$2,$4);};
```

5) Simple unary nodes.

LDF syntax: <NT> ::= sep1 <NT1> sep2

Will produce:

```
NT : tok43 NT1 tok23
         {$$=nun(code_NT,$2);};
```

6) Alternative unary nodes.

LDF syntax: ALT<NT> ::= <NT1> | <NT2> | ... | <NTi> ;

```
Will produce: NT : NT1 {$$=nun(code_NT,$1);}
    | NT2 {$$=nun(code_NT,$1);}
    | ...
    | NTi {$$=nun(code_NT,$1);}
```

7) Unary nodes without constructors.

LDF syntax: SC<NT> ::= <NT1> | <NT2> | ... | <NTi> ;

Will produce:

```
NT : NT1 {$$=$1;}
    | NT2 {$$=$1;}
    | ...
    | NTi {$$=$1;}
```

8) Zeroary nodes.

LDF syntax: <NT> ::= sep1;

Will produce:

```
NT : tok34 {$$=nzer(code_sep1);}
```

9) Generic nodes.

LDF syntax: GEN<name> ::= "[a_zA_Z][a_zA_Z0_9]*";

Will produce:

```
NT : tok31 {$$=constrgen(code_name,yytext,yyleng);}
```

10) Comment nodes.

LDF syntax: GEN<comment> ::= "[a_zA_Z][a_zA_Z0_9]*";

Will produce:

```
NT : tok31 {$$=constrcomm(code_comment,yytext,yyleng);}
```

11) Ident nodes.

LDF syntax: GEN<ident> ::= "[a_zA_Z][a_zA_Z0_9]*";

Will produce:

```
NT : tok31 {$$=constrident(code_ident,yytext,yyleng);}
```

12) Lists nodes.

LDF syntax: L<NT> ::= <NT1>*sep1 ;

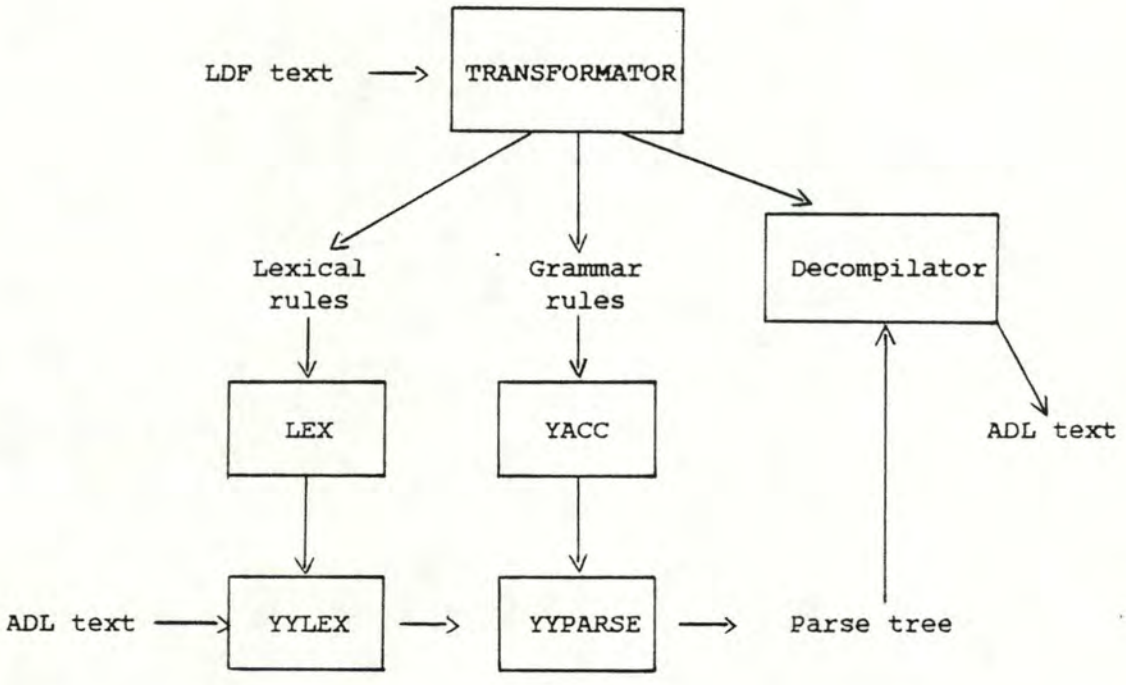
Will produce:

```
NT :    {$$=nil;}
    |   NT tok34 NT1 {$$=clien($1,$3);}
    |   error {* actions associated with the error treatment *}
```

6.3. Conclusions.

The transformator generates all the necessary informations so that the parser can be produced. The specification files generated by the transformator can be found in the appendix. So the tools we have just described allow us to simply write a LDF text of the ADL formalism. Provided that the defined grammar is unambiguous, we obtain a complete parser that performs the lexical and syntactical analysis of an ADL program text and it produces a parse tree. The tool obtained performs other tasks that won't be explained here (like, for example, the decompilation working on the parse tree); more details can be found in [BUYS.85a] and [BUYS.85b].

The whole process can be graphically represented by the following figure.



CHAPTER 6: DESIGN AND IMPLEMENTATION OF THE SEMANTIC ANALYSIS

INTRODUCTION

In a general point of view, the activity of an analyser can be decomposed into three major functions:

1. the lexical analysis, 'scanning' and 'screening' the input stream to recognize the tokens of the language
2. the syntactic analysis, matching the structures present in the input stream with the syntax of the language
3. the semantic analysis.

In our work, each part has been treated separately. However since we use a deterministic method of syntax analysis (an LALR(1) analyser), we could have considered the syntax as a skeleton on which to hang semantics. This would make it possible to execute semantic routines during the syntax analysis process, thus saving a pass of the parse tree. Therefore our option is not the best way to achieve a most efficient analyser, but this clear separation of concerns allows us to use the toolkits available (chapter 5) and to ease several problems faced by the analyser and the text generator. Besides , since we are in an experimental phase, we focus on the simplicity, the readability, the understanding and the correctness of the compiler functions prior to the concerns of time and memory space performance and efficiency.

In the previous chapter we have seen how the syntax of the language allows us to analyse the structure of a program and build its parse tree. This chapter presents the choices which were made in the design and the consequent implementation of the the semantic analyser. The semantic analyser accepts as input a parse tree and produces a symbol table, a declaration table and other data collections useful to the code generator.

The first part concentrates on the various data tables produced by the semantic analyser. It gives the structure of the tables. Moreover it presents the list of routines provided to accede to and manipulate the different tables.

The second part gives the principles which have been adopted for the semantic analysis and an idea of how they have been implemented, using one or the other data table described in the first section.

1. THE TABLES OF THE SEMANTIC ANALYSER

A compiler needs to collect and use information about all the data objects appearing in the source program. For instance, the semantic analyser has to know the type of a variable, the size of an array, the number of arguments of a function and their type, and so forth. This collecting of information is accomplished by means of a symbol table. In the abstract, a symbol table entry is composed of two fields, a name field and an information field.

However, the ADL analyser has to collect such a variety of supplementary information, especially about the objects of the database, that a single data table appeared to be quite insufficient for storing all the information relevant to the code generator. Hence, the semantic analyser does not only produce a symbol table, but also a declaration table for the data items, a record table for the undetermined reference variables and an access key table. For each of them we must be able to :

- determine whether a given object is in the table
- add a new entry to the table
- accede to the information associated to a given object.

The primary issues in designing these tables are the format of the entries as well as the access method. So for each of the data tables, this section first describes the data structure and then lists the functions provided to accede to that structure.

1.1. The Symbol Table

The symbol table collects the relevant information about all the names figuring in the declaration part of the ADL program. It stores also information about the record types and the access path types used in the algorithm.

1.1.1. Structure of the symbol table

An entry of the symbol table consists of :

- the string of characters that denotes the name of the defined object
- the length of the string
- the level number
- the class value
- the reference to the declaration tree
- the code of the object referred to by the name
- the reference to the declaration table.

The level number is highly bound to the way the semantic analyser works. As the analyser moves sequentially downwards through the declaration

part of the parse tree, producing a symbol table entry for each defined name, the order of the names in the table corresponds to their order of appearance in the declarations. This property of the symbol table is used for the implementation of group variables. Consider a variable whose type is GROUP and which contains several fields. If the level number of the entry which denotes the variable is i, then the level number of the entries denoting its fields is i+1. Besides, the level number of an entry which denotes a variable not contained in any other variable is defined to be 1. Finally the level number of all entries which do not denote a variable is also 1.

The class value of the name determines the type of the object the name refers to. The class value is either some type of database object (algorithm, record_type or access_path_type) or some type of a variable (boolean, string, real, integer, group, items_of, ref_name, ref_record or any non standard type declared within the program).

To be very precise, the reference to the declaration tree constitutes the reference, within the declaration part of the parse tree, of the generic node which denotes the name of the symbol table entry. Indeed, as each object used in the program has to be defined in the declaration part, there is a parse subtree containing the declaration of the generic. Nevertheless, note that record types and path types of the database schema are not defined in an ADL program; their tree reference is assumed to be -1.

Only objects of the database the ADL program is working on, do possess a code. Therefore the field of the symbol table entry which specifies the code of the declared name has only a value if the name refers to an algorithm, a record type or an access path type listed in the conceptual schema. The default value is assumed to be -1. The declaration table is produced to store some relevant information about the data items of a record type used in the program. Thus the reference to a declaration table entry contains the index value of the first data item of the record type the symbol table entry is referring to. It follows that this reference only has a value if the name denotes a record type, a variable ITEMS_OF which is defined to receive the item values of some instance of a particular record type or a reference variable dedicated to one particular record type (REF_NAME). The default value is again assumed to be -1.

1.1.2. Access routines

There are eight routines that work on the symbol table; all but one are simple consultations.

1. SB_CREATE(str,inf): allows one to create an entry in the symbol table.

Inputs:
- str: contains the string of characters which denotes a name
- inf: contains the address of a data structure which specifies :
  - the length of 'str'

- the level number
- the class of the object the name refers to
- the reference of the generic 'str' in the declaration subtree
- the code of the object the name refers to
- the reference to a declaration table entry.

Output :

The output is the reference of the created entry or a 'null' value if there is a table overflow.

2. FIND(ref,str,len) allows one to get the reference of the entry of a particular name.

Inputs :

- ref: contains the reference of the starting entry in the symbol table
- str: contains the string of characters which denotes the name to search for in the table
- leng: contains the length of 'str'.

Output :

The output is the reference of the entry in the symbol table if the search succeeds, otherwise the output is a 'null' value.

3. FINDNEXT(ref,class) allows one to get the reference of the next entry in the symbol table which denotes an object of a particular class.

Inputs :

- ref: contains the reference of the starting entry in the symbol table
- class: contains the class value which denotes the type of the object to find in the table.

Output :

The output is the reference of the entry in the symbol table if the search succeeds, otherwise the output is a 'null' value.

4. GET_NAME(ref,str) allows one to get the name of an entry of the symbol table.

Input :

- ref: contains the reference of the entry in the symbol table.

Output :

- str: contains the string of characters which denotes the name of the entry.

The output is the length of 'str' and equals 0 if the input 'ref' was invalid.

5. GET_CLASS(ref) allows one to get the class value of an entry.

Input :
- ref: contains the reference of the entry in the symbol table.

Output :
The output is the class value of the specified symbol table entry or a 'null' value if the input was invalid.

6. GET_TREE(ref) allows one to get the reference of the generic node in the declaration subtree.

Input :
- ref: contains the reference of the entry in the symbol table.

Output :
The output is the reference to the declaration tree of the specified symbol table entry or a 'null' value if the input was invalid.

7. GET_CODE(ref) allows one to get the code of an entry.

Input :
- ref: contains the reference of the entry in the symbol table.

Output :
The output is the code value of the specified symbol table entry or a 'null' value if the input parameter was invalid.

8. GET_DECL(ref) allows one to get the reference to the declaration table.

Input :
- ref: contains the reference of the entry in the symbol table.

Output :
The output is the reference to a declaration table entry of the specified symbol table entry or a 'null' value if the input parameter was invalid.

1.2. The Declaration Table

The declaration table called DECL_TAB collects the relevant information about the data items of all the record types which are used in the ADL program. For each record type, it lists the items in the sequential order of their definition in the database.

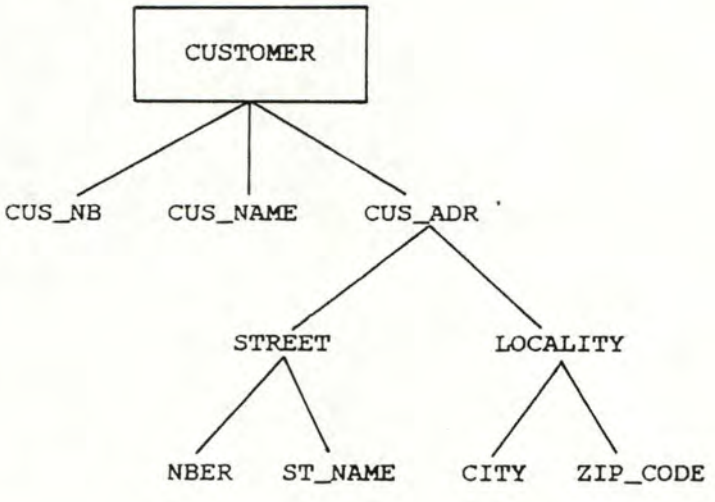
1.2.1. Structure of the declaration table

An entry of the declaration table consists of :

- the string of characters that denotes the name of the item
- the length of the string
- the level number
- the repetitivity of the item the name refers to
- the type of the item the name refers to
- the maximum size of the data item values
- the number of decimals for a numeric data item.

The level number is again highly related to the way the semantic analyser fills up the table. It has the same meaning as for the symbol table, with the only difference that it helps here to implement decomposable data items. Moreover, the level number of the first item of a given record type is assumed to be 0 in order to mark the beginning of a new list of data items.

Example :



NAME	LEVEL
CUS_NB	0
CUS_NAME	1
CUS_ADR	1
STREET	2
NBER	3
ST_NAME	3
LOCALITY	2
CITY	3
ZIP_CODE	3

Each item of the database is characterized by a number indicating its repetitivity. This number is defined to be 0 if the data item is non repetitive, otherwise it gives the fixed repetitivity of the item.

In the conceptual schema, a data item of a record type is assumed to be either decomposable, numeric or a string of characters. Thus the type of a declaration table entry has these three possible values.

The maximum size of the item values specifies the maximum length of the values which may be assigned to that data item.

Finally, for the items which are of type numeric, a declaration table entry indicates the number of decimals (i.e. the number of digits after the decimal point).

1.2.2. Access and manipulation routines

The routines which work on the declaration table are :

1. DT_CREATE(str,length,ditem) allows one to create an entry in the symbol table.

Inputs :
- str: contains the string of characters which denotes the name
- length: contains the length of 'str'
- ditem: gives the address of a data structure containing the information to be stored into the table; it specifies :
  - the level number of the item in the decomposition
  - the repetitivity
  - the type
  - the maximum size of a value of the item
  - the number of decimals of the item values if the item is numeric.

Output :
The output is the reference of the created entry or a 'null' value if there is a table overflow.

2. GET_ITEMS(str,length,ditem) allows one to get an entry in the symbol table by reference.

Input :
- ref: contains the reference of the entry in the declaration table.

Output :
- str: contains the string of characters that denotes the name of the entry
- ditem is the same as above.

The output is the length of the name of the entry in the declaration table.

3. GET_NEXT(ref,str,plen,ditem) allows one to get the following entry of a particular declaration table entry.

Input :
- ref: contains the reference of the previous entry in the declaration table.

Outputs :
- str: contains the string of characters that denotes the name of the entry
- plen: points to the variable containing the length of 'str'
- ditem is the same as above.

The output is the reference of the next entry if there is any, otherwise it is a 'null' value.

4. GET_DATA(ref) allows one to get the data stored about an item in the declaration table.

Input :
- ref: contains the reference of the entry in the declaration table.

Output :
The output is a structure of five fields which specifies :
- the level number
- the repetitivity
- the type
- the length of the maximum size of the values
- the number of decimals of the item which is referred to by the declaration table entry.

5. FIND_STR(ref,str,length) allows one to get the reference of the entry of a particular name.

Inputs :
- ref: contains the reference of the starting entry in the declaration table
- str: contains the string of characters which denotes the name to search for in the table
- leng: contains the length of 'str'.

Output :
The output is the reference of the entry in the declaration table if the search succeeds, otherwise the output is a 'null' value.

6. FIND_IT(ref,str,length) allows one to get the reference of an entry of a particular name that refers to a simple item.

Inputs :
- ref, str and leng are the same as above.

Output :
The output is the reference of the entry in the declaration table if the search succeeds, otherwise the output is a 'null' value.

7. FREE_ENT( ) allows one to get the index value of the first empty entry in the declaration table.

Input : /

Output :
The output is the reference to the first available empty space in the table.

8. TOT_LEN(ref) allows one to get the total length of a record type (i.e. the sum of the maximum sizes of its data item values).

Input :
- ref: contains the reference in the declaration table of the first item of the given record type.

Output :
The output is the total length of the record type.

Remark :

Note that all the routines which work on the declaration table and require the value of a reference do have a precondition: the reference passed as input is assumed to be valid, in other words to point to a filled entry of the declaration table.

1.3. The Record Type Table

The record type table is produced to supply the code generator with the list of record types which may be the target of variables defined "ref of RECORD". This table, called REC_TAB, stores the record type names associated to the reference variable.

1.3.1. Structure of REC_TAB

An entry of the record table is composed of :

- the string of characters which denotes the name of the reference variable
- the string of characters which denotes the name of the record type.

Note that both strings are limited to the size of database names, that's to say a maximum size of twelve characters.

1.3.2. Access routines

The two routines manipulating the record type table are :

1. RT_NEXT(str,ref,str_rec,leng) which allows one to accede to the record type following a particular entry and referenced by a certain reference variable.

Inputs :
- str: contains the string of characters which denotes the reference variable
- ref: contains the reference of the previous element in the table or a 'null' value if one wants to get the first record type linked to the reference variable.

Outputs :
- str_rec: contains the string of characters which denotes the record type
- leng: gives the address of a variable containing the length of 'str_rec'.

The output is the reference of the acceded entry or a 'null' value if there is no other record type associated to the reference variable specified in 'str'.

2. RT_CREATE(str,str_rec) which allows one to create an entry in the record type table.

Inputs :

- str: contains the string of characters which denotes the reference variable
- str_rec: contains the string of characters which denotes the record type.

Outputs :

/

1.4. Table of the Access Keys

The access key table, called IKO_TAB, is introduced by the semantic analyser to provide the COBOL generating process with the code value of the keys used in the access requests of the ADL program. As we pointed out already (see chapter 4: description of an IKO), the access key is characterized by its code but doesn't have a name. It is composed of one or more data items which, in return, may be the components of more than one access key.

On one hand this many-to-many relationship between a data item and an IKO made it impossible to store the access keys into the symbol table using the name of their component items to reference them. On the other hand the generator needs to know the codes of the used access keys and it does not pass the tree in a top-down left-right order. So we are forced to save the codes in a separate table and associate to them some piece of information which guarantees to identify these codes . These are the reasons for the structure that has been adopted.

1.4.1. Structure of the access key table

Each entry of the access key table consists of :

- the line number of the for_statement in the ADL text
- the string of characters which denotes the name of the reference variable
- the code value of the access key
- the set of references of the data items forming the access key.

This structure shows that we assume an access key to be identified by the line number of the for_statement in which the key is used and by the name of the used loop variable.

The string denoting the loop variable is limited to twelve characters, the maximum size of database names, and currently the maximum number of components of an access key is assumed to be five.

1.4.2. Access routines

We defined only three routines to work on the access key table :

1. GET_IKO(lnb,str) allows one to get the code of the access key used in the statement identified by the line number and the reference variable.

Inputs :
- lnb: contains the line number of the for_statement
- str: contains the string which denotes the name of the reference variable.

Output :

The output is the code of the access key or a 'null' value if the search for the key fails.

2. NEXT_IKO(ival,pkey) allows one to sequentially accede to the elements of the key table and to get the stored information.

Input :

- ival: contains the index value of the entry which has been acceded previously. A 'null' value causes an access to the first entry of the table.

Output :

- pkey: references a data structure which contains :
  - the code of the access key within the database schema
  - the reference of the components of the access key, in the order they constitute this IKO; this reference is the address of the related data item within the declaration table.

The output is the reference to the next entry, if there is yet another key listed in the table. Otherwise the output is a 'null' value.

3. PUT_IKO(lnb,str,code) allows one to create an entry in the access key table.

Input :

- lnb: contains the line number of the for_statement
- str: contains the string which denotes the name of the reference variable
- code: contains the code of the access key identified by the first two parameters.

Output :

/

2. THE SEMANTIC ANALYSIS

In the definition of a language, the syntax is concerned with the form of things and the semantics with their meaning. Thus the term of semantic analysis is applied to the determination of the underlying logic and sense of the algorithmic structures defined by the syntax of the language. It is concerned with the control of the meaning of the various expressions and statements. It checks context_dependent features impossible to be integrated in the formalized definition of the syntax, as this formalization uses a context_free grammar.

Semantic analysis can be done during or after the syntactic analysis phase. As we mentioned already, the semantic analyser was designed as a completely separate phase, and its purpose is multiple :

- First the semantic analysis is used to check the correctness of any expression recognized by the parser. It checks that the arguments of an expression have been defined and it determines their type. It even evaluates the type of intermediate results in order to judge if the application of the operators is legal.

- For any statement specifying some action on the database, the semantic analyser studies its meaning in order to be able to decide if the statement logically has a sense. In the limits of its possibilities it even foresees if the required action does not endanger the consistency of the database.

- Finally, the semantic analysis phase is designed to facilitate the work of the code generator. For this purpose it gathers any piece of relevant information about the different types of database objects, thus producing the data tables described in the previous section.

This section is dedicated to these different functions of the semantic analyser. It briefly recalls the semantic principles which have been adopted (see chapter 3) and describes the way they have been implemented, with regard to some restrictions introduced for complexity reasons. This is done by first explaining the handling of the variables, a central element of an ADL program; then we see the other objects which have to be declared in a program. The analysis of the statement part begins with the presentation of two chief expressions: the arithmetic expression and the database object set, which is the expression of the database access requests. Sets of similar statements are discussed before we terminate with some remarks about the semantic errors and error recovery.

2.1. Managing Variables in Declarations and in Statements

2.1.1. Semantics

Each variable or name used in the program must be declared and the identifier has to be unique.

2.1.2. In the declaration part

Passing through the declaration subtree, the semantic analyser fills the symbol table and the declaration table. For each variable declaration encountered, an entry is created in the symbol table with all the relevant information (see section 1.1).

Of course, we had to find a mechanism to determine the level number of a name contained in an other variable. Since the entries in the symbol table are created in the order of appearance of the corresponding names, we managed the problem of group variables by implementing a global variable which is supposed to contain the level number of the presently analysed field identifier. This variable is incremented at the encounter of a group variable and decremented at the end of its structure declaration. This simple mechanism guarantees that at any time we know the level we are working at.

Another problem we had to cope with, as far as the group variables are concerned, was to check that the identifiers on a same level in the structure are unique. To handle this semantic test, we needed to save the reference of the last field in the symbol table before going down one level. This is necessary as the routine FIND (whose job is to look for the entry that corresponds to a given name) always requires two arguments :

- the name to be searched out
- the reference of the entry from which the search has to start.

This routine starts from the entry indicated by the second parameter and searches out an entry which contains i) a name matching the first parameter and ii) a level number equal to one plus the level number of the starting entry. It will stop if it finds such an entry (successful search), or if it encounters an entry that contains the same level number as the starting entry or, obviously, if it reaches the end of the symbol table (unsuccessful search). The storage of the references of variables and field names which are structures required the implementation of a stack. Before going down one level, the analyser pushes the reference onto the stack and, at the end of the analysis of a substructure, it pops the last reference off the stack. Of course the check is positive if the routine FIND ends with an unsuccessful search.

Remark :
To check the uniqueness of a name that has the level number one, you

just need to put a 'null' value in the second argument of the routine FIND.

Variables may be declared of one of the standard types, simple or structured, or of a type defined earlier in the program. In this last case the semantic analyser checks that the identifier denoting the non standard type, has been declared in the type declarations of the program. In the symbol table, this kind of variables are listed as being of class TYPE.

Finally, the declaration of variables defined to receive the data item values of a record (ITEMS_OF) or to reference a record (REF_NAME) of a given type, semantically requires the specified record type to exist in the schema this program is defined to work on. Hence the semantic analyser accedes to the conceptual schema in order to search out the record type. If it is successful, it adds the record type with its code value to the symbol table and stores its data items into the declaration table, in the sequential declaration order. This way of listing all the items of the used record types assures us that each name used in the statement part of the program has to be present in one of both tables. However we have to do an exception for the path type names (they are only known in the statement part), but fortunately the contexts in which they may be used are so restricted that we can foresee a special treatment at the right places.

Note that the declaration of a reference variable REF_RECORD which is not dedicated to a special record type is not handled particularly. Indeed the list of record types it may reference, can only be determined when we analyse the access request(s) where it is used. At that moment, if the record type is not yet listed in the symbol table, it is inserted and its data items are also listed in the declaration table. Thus all the objects that may be addressed at some moment in the statement part are sure to be present in either the symbol or the declaration table.

2.1.3. In the statement part

While analysing the statement part the parser consults the symbol table at each variable or name encountered. It verifies that each identifier has previously been declared, either explicitly in the program or implicitly via the schema description stored in the database of the system.

Here, group variables (they are denoted by hierarchical variables) are analysed by a mechanism similar to that described above for the declaration part. The token that signifies a jump to a lower level of the structure is a period between two names. A variable of the form XXX.YYY denotes the field 'YYY' of the structured variable 'XXX'. It follows that the name 'YYY' has to be searched out in the list of names which are just after 'XXX' in the symbol table and have a level number equal to one plus the level number of 'XXX'. Consequently, we simply need to use the routine FIND with the reference of 'XXX' in the symbol table and to test if the search is successful. Thus we assure that the qualification of a given field of a complex structure is done correctly.

A special treatment is necessary if the structure refers to a data item of some record type, that's to say when the variable is of the form :

‹name›. ... where the name denotes a variable of type ITEMS_OF or

(‹name›). ... where the name denotes a variable of type REF_NAME or REF_RECORD.

The check is quite similar by using the routine FIND_STR, as the declaration table entry also includes a level number. The only difference is that the starting point for the first field is determined by the value found in the 'decl' of the symbol table entry that respectively denotes the reference or items variable. Moreover, the semantic analyser checks that the variable used has been defined for the right record type, if the latter is stated explicitly in the statement, as it is the case in ‹for›, ‹create› and ‹modify› statements.

The search of the data items appearing in these statements is somewhat different, as the syntax only allows to specify the terminal item and not the complete hierarchical qualification. In fact, the leaves of the record structure have to be searched sequentially which is realized by the routine FIND_IT. This routine asks for the same two parameters, but it does not require that the name it looks for has a level number equal to one plus the level number of the starting entry which is specified by the second parameter. The string has only to denote a non decomposable item of the concerned record type.

Finally the semantic analyser has to do some additional testing if the variable is subscripted. Indeed it checks that the variable has been declared to be an array and that the subscript matches the dimension of the defined array. If the index is an integer value, the analyser tests if it lies within the range specified in the declaration. Moreover, when the subscripted variable happens to denote a repetitive data item of a record type, it logically follows that the subscript is one-dimensional and that the item has to be repetitive.

Remark on the variables declared "ref of RECORD"

A reference variable is declared "ref of RECORD" if it is used in an access within an access path type that has more than one record type as a possible target. The type of the records referenced by such a variable is unknown a priori; the determination of the type of the records treated in a given portion of an algorithm (for instance in a conditional branch) depends on the context. This means that the semantic analyser must interprete the alternative statement in order to check that the variable expressions address data items of the assumed type of records. It is also impossible to have a conditional branch which still admits more than one defined record type (case of a simple else statement when the logical exclusion has yet different possibilities).

The possibility of conditional expressions which do not explicitly mention the case that is considered, as they are based on the logical exclusion principle, makes the determination of the context very complicated. So we decided to do no testing whatsoever if the reference variable is not dedicated to a single record type. The semantic analyser only produces the list of the possible targets of such a reference variable because it is necessary to the code generating process. This option, however, may introduce a problem of ambiguity in the qualification of some data items in the generated COBOL code. We shall come back to this later on.

2.2. Managing Non Standard Data Types

2.2.1. Semantics

Each type that is declared must have a unique name and a type which is used in the declaration of a more general type must be declared beforehand.

2.2.2. Type declaration part

For each type declaration, the semantic analyser consults the symbol table to make sure that there is no previous declaration with the same name. It uses the search routine FIND (described in more detail in section 2.1.2.) giving as starting entry a 'null' value, and it tests if the search fails.

The description of the type structure is analysed exactly as the fields of a declared group variable. Thus an error is encountered if one field is of a non standard type which has not been declared earlier in the type declaration part. At the same time it is guaranteed that the fields at a same level do have unique identifiers.

2.3. Managing the Algorithm Declaration

The name of the algorithm is assumed to reference a module listed in the conceptual schema of the database. A module (see chapter 4) is characterized by its name, of course unique, its code and its type. The latter specifies if it is a predicative or effective algorithm.

The name of the algorithm is the only thing that is declared in the extern declaration part. Thus this name constitutes the only means to determine the schema on which this module is going to work. So the semantic analyser uses the sequential access routine of the GAM (see section 4 of chapter 2) to find the module which matches the specified algorithm name. From there it accedes to the associated database schema (access along the path type S-M-I). An entry in the symbol table is created to store the algorithm name and its code whereas the code of the schema is assigned to a global variable. This code value will be used later on to accede to the record types or path types of the schema.

Of course the semantic analysis is immediately interrupted if the opening of the database of the system fails. A check which could also be implemented is to verify that the found module is an effective algorithm, as the compiler is only supposed to handle effective ADL programs. This test has not yet been implemented, because the description of the modules in the conceptual schema is still incomplete. Indeed, as every function is supposed to be listed as a module in the system, the schema has to include some means for describing the input and output parameters of a module. This state of affairs also determines the way of handling function calls (section 2.9.)

2.4. Managing Arithmetic Expressions

Because of the definition of the syntax, the non terminal ARITHMETIC_EXPRESSION does not only denote pure arithmetic operations. It can also be a database access request, a set to 'null' of a reference variable (NULLREF), any kind of string, variable or function call. But the allowed operators are purely arithmetic, being the addition, substraction, multiplication, division and the power operation.

This generality of the non terminal ARITHMETIC_EXPRESSION makes it necessary to test if the application of the operators is legal according to the type of the arguments. It is semantically required that each operand is numeric, that's to say integer or real.

As this general syntactic structure may be of quite different types and as it is used in multiple contexts we determine the type of the entire expression, thus making it possible for the larger context to decide if it is legal. We evaluate the type gradually with respect to the following semantic rules :

- if the operator is "+", "-" or "*": if the two arguments are integers the result is integer, otherwise it is real;
- if the operator is "/" or "^": the result is assumed to be real;
- if the argument is not numeric, it necessarily forms the expression on its own, thus determining the type. Therefore the expression may be of type NULLREF or DB_OBJECT_SET, of type STRING or BOOLEAN or of any type a variable or a function call may be of. The type of a variable is determined by acceding to the class value of the corresponding symbol table entry or the type of the corresponding declaration table entry.

Remark

The present impossibility to determine the type of a certain function, given the incompleteness of the conceptual schema, we allow the arithmetic expression to be of type UNDEFINED. This default value is always assumed to be a valid type, thus leaving the entire responsability with the programmer.

Note that we consider only INTEGER and REAL. Hence as variables and data items are declared as simply being numeric, we consider the specified number of decimals in order to be able to determine if the quantity is either integer or real. Moreover, different designations for a same type are unified under one single type: for instance a logical value is assumed to be of type boolean.

2.5. Managing Database Access Expressions

2.5.1. Semantics

The syntactic structure which denotes the database access expressions is called DB_OBJECT_SET. It may specify a sequential access, an access within a path type or/and an access by key (see chapter 3). The semantic conditions depend on the type of the access, but the type of the reference variable used for the target records must also be considered.

If this reference variable is of type REF_NAME, it logically has to be associated to the record type which is explicitly named in the syntactic expression of the set of selected records (DB_OBJECT_SET). Of course this record must belong to the schema and all four types of access are allowed.

If the reference variable is of type REF_RECORD, the access expression denotes the string "RECORD" at the place of the record type name. Moreover the access described may only be an access within a path type which, in addition, must be listed in the conceptual schema as having various record types as possible target.

The remaining semantic constraints concern the predicate condition which expresses a path or key condition. The access within a path type is denoted by the syntactic structure RELATION_CONDITION, in which the first terminal specifies a path type name.This path type must belong to the schema and it should link the record types which are addressed by the two reference variables named in the complete access expression:

- the reference variable present in the relation condition determines the type of the origin record (this variable must be of type REF_NAME)
- the variable to which the records satisfying the access condition are assigned to, references the target record type (it is of type REF_NAME) or target record types (it is of type REF_RECORD).

The access by key is expressed by one (a simple access key) or by several (a decomposable access key) belonging conditions. Such a conditional expression has to denote one or the component of the key, this component being a simple data item of the concerned record type. Moreover it is required that the component data item conditions are given in the order in which they appear in the declaration of the acces key. We even assume that this order corresponds also to the order of the data items within the record type declaration. Evidently the value specified in a belonging condition should be of simple type and compatible with the type of the data item.

Finally the semantics asks that, in the case of an access by key within an access path, the access path condition is specified in the first place. The access key has to be defined as having this access path as a reference set and, from another point of view, it may not be used without the corresponding access path condition. Note also that we logically exclude a path type without any name. We don't think that this introduces any major restriction and the tracing within the conceptual schema of a

given access path type is facilitated. Furthermore, this possibility would add to the complexity of the testing of non dedicated reference variables.

2.5.2. In for and assign statements

To achieve the semantic tests based on the principles described above, we defined two global variables, called 'acckey' and 'accpath', which respectively indicate if an access key or an access path condition has been specified in the predicate. Besides the second variable allows us to check that an access request does not specify two access path conditions, a semantic violation. Yet another test, not directly mentioned in the semantics: we check that, if a predicate condition is enclosed between parentheses, it contains at least two separate conditions and that these parentheses enclose the whole predicate expression. In other words, a predicate within parentheses cannot be one conditional expression of a more general predicate. We introduced this restriction, which in fact isn't one, to standardize the writing of the access conditions. Thus expressions like the following are assumed incorrect :

```
CUS := CUSTOMER(( :NUM = 123))
ORD := ORDER(CC:CUS) and ((:DAY = 01) and (:MONTH = 07)
                          and (:YEAR = 85))
```

The corresponding correct expressions would be :

```
CUS := CUSTOMER( :NUM = 123)
ORD := ORDER((CC:CUS) and (:DAY = 01) and (:MONTH = 07)
                            and (:YEAR = 85))
```

which seem to us more understandable, grouping all the conditions together.

Moreover, we refuse negated conditions which are allowed by the syntax of a predicate, as they can't be translated directly into one of the access functions offered by the GAM (see section 4 of chapter 2). Hence they do not belong to an effective algorithm.

To test the existence of a record or path type that is known by its name, the semantic analyser consults the conceptual schema. It accedes to the record or path types sequentially within the one-to-many access path type which leads from the record type SCHEMA to the desired type of record (see chapter 4). The iteration stops if a record matching the name specified in the program is found or if the set of path or record types of the schema is exhausted. In the case of a successful search, the analyser gets the code value of the database object and creates an entry in the symbol table with the information it has got. This way, it avoids multiple searches in the database for a same path or record type. In addition, the code value can be used as an access key in the other necessary tests, like the verification of the type of the origin and target of an access path or the control of an access key of a record type.

To test the origin and target of an access path type (see schema description in chapter 4), we use the path type M-P defined between the record types PATH-TYPE and MEMBER. Depending on the role of the member, the

RECORD-TYPE being associated via the path type R-M-I to the acceded member is either an origin or target of the considered path type. Thus the specified record type names can be checked. This is also the moment to determine the list of the record types which may be target of a general reference variable. So, if the access request specifies a record variable of type REF_RECORD, all the target members of the access path type are inserted in the data structure REC_TAB (section 1.3.). For the record types which are still not listed in the symbol table, a symbol table entry is created and their associated data items are stored into the declaration table.

An access key however is checked by taking the path type CONCERNS which links a RECORD-TYPE to its IKOs. For the IKOs declared to be an access key, we get the components associated to it by moving along the path type IKO-C. Depending on the value of SEC-NUM, the associated data item is the i-th component of the access key. This iterative check is continued until finding an access key that is composed of the data items specified in the belonging conditions.

Finally, we had to take into account a restriction introduced by the access functions of the GAM: the access by key only allows the equal sign as a test operator. So the semantic analyser makes sure that this constraint is respected.

2.6. Managing For Statements

2.6.1. Semantics

The for_statement may have two completely different meanings, depending on the type of the collection expression.

If this syntactic structure denotes a range expression, the loop variable must be of type integer and the range expressions must also be either a variable of type integer or an integer value.

But if the non terminal COLLECTION_EXPRESSION is formed of a database access expression (DB_OBJECT_SET), the loop variable has to be a reference variable. For the semantic constraints of the access request, please refer to the previous section (section 2.5.).

2.6.2. Implementation

Although the testing is quite different depending on the kind of collection expression, the semantic analyser always has to check the type of a variable. This is rather direct, as it stores the class of a variable in the symbol table when analysing its declaration. Indeed it only has to get the class value of the entry which denotes the variable. The test that a range expression, if it is not a variable, is an integer value is even more direct, as the type of the path tree node specifies this.

2.7. Managing Next and Exit Statements

2.7.1. Semantics

Both statements determine a break of a loop execution, one asking the termination and the other the abortion of the execution of the body of the loop. If these statements specify a name, this terminal logically must denote a loop control variable which is still active. We mean by that, that the variable has to be used in a 'for' statement which is not yet terminated.

2.7.2. Implementation

Whereas the check that the name is that of a reference variable is achieved without any problem, the test of it being an active loop variable requires the implementation of a stack. This stack stores at any moment the loop variables of started and not yet finished 'for' statements. The top element of the stack denotes the variable of the innermost loop. By consulting the list of variables stored in the stack, the semantic analyser is able to execute the test. Obviously this stack has to be updated at each encountered begin or end of a 'for' statement.

2.8. Managing Conditional Statements

With the term of conditional statement we designate the 'if' statement, the 'if_then_else' statement as well as the 'while' statement. They have been grouped together because each of them includes the non terminal GENERAL_EXPRESSION, the only syntactic structure that has to be checked semantically.

2.8.1. Semantics

The syntax of the ADL language defines a general expression as being a simple test expression or a logical combination of several test expressions. As the syntax assures the exclusive use of legal logical operators, the semantic analysis is only concerned with the correctness of the individual test expressions. For these syntactic structures it is semantically necessary that the quantities being compared are of compatible types. The rules we adopted exclude the comparison of complex data structures. Are considered as valid tests :

integer or real with integer or real
string with string
boolean with boolean (a logical value is assumed to be boolean)
nullref, ref_name or ref_record with nullref, ref_name or ref_record.

In the last two types of comparisons the test operator has to be either the equal or the non equal sign ("=" or "<>"). Besides, as an arithmetic expression is of an undefined type if it is formed of a single function call (see section 2.4.), we assume comparisons with the default type to be semantically valid.

2.8.2. Implementation

The two quantities compared in a test expression are represented by the syntactic structure of an arithmetic expression. The semantic analysis of such an expression (see section 2.4.) provides its type. Consequently the only task the analyser has to accomplish is to apply the rules stated above, once it knows the type of both expressions involved.

2.9. Managing Assign Statements

2.9.1. Semantics

This kind of statement represents the assignment of the quantity specified on the right side of the assign symbol to the variable on the left side. It semantically follows that both are of compatible types if not of the same type. Thus, are considered as valid assign statements, those where the variable and the assign expression are of the same type as well as the assignation of an integer to a real and the assignation of a null reference or a database object to a variable declared "ref of ...". Nevertheless, the assignation of complex data structures, of group variables, arrays or variables of a non standard data type is not tolerated. As an assign expression may denote a function call, its type may be undefined and the assign statement is assumed valid, provided that the variable is of simple type.

A case to be considered particularly is an assign expression that denotes a variable of the form (<name>).ITEMS. This expression designates all the item values of the record currently referenced by the record variable named in the parenthesis. Semantically the variable is required to be of type ITEMS_OF and to be defined for the same record type as the reference variable. In general, if an assignation concerns two dedicated reference variables, they have to be defined for the same record type.

Finally the semantics of the language requires that the assign statement does not represent the modification of a data item value of some record. In concrete terms this means that the variable of the statement must not denote a data item of a database record type.

2.9.2. Implementation

To achieve all these verifications, the semantic analyser first determines the type of both, the variable and the assign expression. The type determination of the assign expression is not too difficult being either a general expression, in which case the type is boolean, or an arithmetic expression, in which case the type is provided by the analysis of this non terminal (section 2.4.). After that, the semantic analysis consists of applying the various criteria stated above, consulting the syntax tree if necessary in order to accomplish the necessary semantic checks.

The parse tree must be used to check that two reference variables of type REF_NAME or that a variable of type REF_NAME and a variable of type ITEMS_OF are declared for a same record type. In this case we use the reference to the declaration subtree stored in the symbol table entry of each variable. From this tree node we manage without any difficulty to access the terminal node which represents the record type name the variable is declared for. A final string comparison achieves the semantic check.

2.10. Managing Call and Return Statements

2.10.1. Semantics

A call statement represents a function invocation whereas the return statement represents an order to get out of a function execution. Both statements may specify a list of parameters which are defined by the syntax to be variables.

Of course the semantics asks for the function named in the call statement to be defined in the system, meaning that it must be listed in the conceptual schema. In addition, the variables passed as arguments in the function call should correspond to the parameters the function is defined to work on: both should be of identical types and given in the same order. In the return statement the variables should represent output parameters of the function.

2.10.2. Implementation

No semantic test whatever has been implemented. The reason, mentioned already in section 2.3 of this chapter, is that the conceptual schema is incomplete, as it only lists the various modules with their name and type. The model foresees no way to store some information about the parameters nor about the relationship between different modules. This part is still in a very experimental phase.

2.11. Managing Database Modification Statements

In this section we shall consider the creation, modification or deletion of a record of the database. As the syntax of the delete statements is quite different from the two other statement types, we shall present them separately.

2.11.1. Delete statements

The problem of deletion is a most difficult question in the domain of database manipulations. After long discussions, the complexity of a meaningful semantic check made us leave the whole responsability with the programmer and the DBMS. Thus the only thing done by the semantic analysis is to test that the terminal of the statement expression denotes some declared reference variable of the program, dedicated to one single record type. A general reference variable is not allowed, as the GAM function asks for the code of the record type concerned and this one is only known indirectly through the reference variable.

2.11.2. Create and modify statements

2.11.2.1. Semantics

Although the syntax differs in some points, most semantic criteria apply to both statements. Indeed it is the syntax which largely controls the constraints that are particular to each of the statements.

The first terminal node in each syntactic structure should denote a reference variable defined for the record type which is named explicitly in the create statement. Like for the delete statement, in the modify statement the record type name is only known indirectly via this reference variable; consequently we refuse the possibility of a non dedicated reference variable for all database modification statements in general. In our opinion, this is not a severe restriction as the programmer may define a reference variable for each possible target record type, besides the general reference variable necessary for the access within the multiple target path type. Depending on the context, he has only to use one or several dedicated reference variables (correctly initialized) for the database manipulation.

Each item condition specified in the statement condition must specify a data item of the considered record type. The data item has to be of a simple type (in other words not decomposable) and the arithmetic expression representing the data item value must be of a simple data type compatible with that of the item. Moreover a correct item condition does not specify an access path name. This is still allowed by the syntax of the expression. Finally, it is semantically assumed that the item conditions denote the data items in the order they have been declared within the record type.

This does not exclude the possibility of skipping one or the other data item.

An attach condition must specify the name of an access path type (we assume that every path type of the schema has a name) and must denote a reference variable that has been defined for the record type origin of the access path type. Evidently the created or modified record must be of the type of the target of the access path type.

Only allowed in a modify statement, a detach condition must fulfil the same semantic criteria, besides the fact that the integer value has to be 0.

Finally note that the conditional expression of the creation of a record does not require the item conditions to be expressed before the attach condition(s).

2.11.2.2. Implementation

The semantic analysis of the reference variable is analogous to the one described in the second paragraph of section 2.8.2. Furthermore the check of the access path type, required in both the attach and detach condition, is explained in section 2.7.2. Finally there remains the testing of the item conditions; this one is achieved by consulting the declaration table. As the terminal data items of a record type are assumed to be sequentially stated by the conditions, we use the access routine FIND_IT. The parameters of this function are described in the section presenting the declaration table (section 1.2.) and its mechanism is explained in the description of the handling of variables in the statement part (section 2.1.3.).

2.12. Semantic Errors and Error Recovery

The primary sources of semantic errors are undeclared names and type incompatibilities.To these we must add the different tests of consistency between the description of the database schema and the database manipulations formulated in the clauses of the program. Finally multiple declarations of a same identifier and incorrect or missing subscripts of array variables are reported.

The error detection however, is only one part of the tasks a compiler has to accomplish. Besides looking through a source program for possible errors, the compiler should, if the program is incorrect, give all useful information for correcting it. At last, if the source text is correct, it is translated into an equivalent program coded in a lower-level language. So the need for a good communication with the user is not to be neglected. Clear error messages, diagnostics as well as some debugging facility are common requirements for a compiler.

As we are in an experimental phase, the ADL compiler is being considered as a prototype, we didn't emphasize the dialogue with the user. The semantic analyser has a defined set of error messages and prints the adequate message for the encountered error. Moreover we didn't work out any sophisticated error recovery mechanisms. Although a same error will be reported each time it is encountered, we chose to simply indicate the erroneous declaration, expression or statement and then ignore it completely. In concrete terms, it means that wrongly declared variables will throughout the algorithm part be looked upon as being undefined, thus causing an error message at each appearance. It means also that the semantic analyser skips the rest of an expression as soon as it detects an error. This rule is valid for the statements too, except for the conditional statements (if and while statements) as a possible error in the general expression does not affect the semantic interpretation of the statements that form the conditional branches or the body of the loop. Instead we do assume that a for_loop which specifies the selection of a defined set of records contains extremely context-dependent actions. Therefore we decided to stop the analysis of the entire for statement as soon as an error is found in the loop control expression. We are well aware of the options being rather drastic, but they have the advantage of largely simplifying the analyser's job. In the present state of experimentation, simplicity and clearness seemed preferable, the main objective being a first understanding and handling of the problem.

CHAPTER 7: THE COBOL GENERATOR: THE GENERATION PRINCIPLES.

1. INTRODUCTION.

This chapter presents the generation principles. That is the generation of the Cobol program corresponding to an ADL program. The generation process uses the parse tree generated at the syntactical analysis and some informations generated at the semantic analysis.

Cobol is a language different in some ways from ADL. A programmation language has always limitations, constraints. The generation of the Cobol program has to take into account some constraints. Some Cobol limitations imply some constraints on the way Cobol program are generated, they will be presented in due time.

Anyway, a Cobol program has a well pre-defined structure with four divisions. Section 2 will present the generation principles for the IDENTIFICATION DIVISION and the ENVIRONMENT DIVISION. Section 3 will present the generation principles for the DATA DIVISION. Section 4 will present the generation principles for the PROCEDURE DIVISION.

The architecture of the generator can be found in the appendix.

2. GENERATION PRINCIPLES FOR THE IDENTIFICATION & ENVIRONMENT DIVISIONS.

The generation principle for those 2 divisions is very simple.

In Cobol, the identification division is used to give some identification parameters of the program. Only one is mandatory: the program identification with the program name. Only that parameter will be generated. The program name is the algorithm name of the ADL program.

The environment division describes the equipment used: hardware and software. Only the CONFIGURATION SECTION is mandatory. It describes the central equipment, that is the computer used for the compilation (SOURCE COMPUTER) and the computer used for the execution (OBJECT COMPUTER). Those parameters are generated with the name of the computer(s); here they are the same: VAX-VMS.

Thus, the generation of these two divisions gives:

```
IDENTIFICATION DIVISION.
************************

PROGRAM-ID. <ADL-algorithm-name>.

ENVIRONMENT DIVISION.
*********************

CONFIGURATION SECTION.

SOURCE-COMPUTER. VAX-VMS.
OBJECT-COMPUTER. VAX-VMS.
```

Remark. The Cobol program generated will be stored in a file. That file has the following name: <ADL-algorithm-name>.CBL. Example, if the algorithm name is ORDER, the Cobol program will be stored in a file name 'ORDER.CBL'.

3. GENERATION PRINCIPLES FOR THE DATA DIVISION.

3.1. Introduction.

The data division gives the description of the data structures. The only section presents is the WORKING-STORAGE SECTION that gives the description of the data stored in the intern memory of the program.

The WORKING-STORAGE SECTION contains different kinds of data. They will be presented in the next section. First, we have the list of the paramaters for the interface with the GAM. Then, we have the generation principles for the data declared explicitly in the ADL program. Section 3.3. will presents those principles.

3.2. Parameters for the interface with the GAM.

The parameters for the interface with the GAM are generated as follows (the general specification of the GAM functions and the description of each parameter can be found in the appendix).

```
* LIST OF THE PARAMETERS FOR THE INTERFACE WITH THE GAM
*-----------------------------------------------------

01  DBSTAT.
    02  FNCODE; PIC 99.
    02  ERRCODE; PIC 99.
01  Z-CODES.
    02  OPCODE; PIC 99; USAGE COMP.
    02  GETCODE; PIC 9.
    02  RECREF; PIC S9(10); USAGE COMP.
    02  KEYCODE; PIC 99; USAGE COMP.
    02  OPERATOR; PIC 9.
01  Z-VALUES.
    02  RECTCODE; PIC 99; USAGE COMP.
    02  Z-VALUE; PIC X( <maxldb> ).
```

***

The generation principles for the following clause needs some explanations, so the Cobol text is interrupted and the next part will be given after the explanations. So the '***' symbol indicates that a comment on the generation principles begins or ends.

Here are the explanations for the Z-VALUES clause. <maxldb> is an integer that represents the maximum lenght of the database records (concatenation of all items of each database record).

As Z-VALUE is used to contain the values of the data item of a given record type and used too to contain the value(s) of the item(s) of an access key, and as these informations don't have the same data type, Z-VALUE needs to be redefined for each database record type and for each access key, with their corresponding data items.

Let's suppose now that, in the current example, we have a database record type and an access key. The Cobol program will be:
***

```
    02  Z-VALUE-CUSTOMER; REDEFINES Z-VALUE.
        03  NAME; PIC X(20).
        03  ADRESSE; PIC X(30).
        03  FILLER; PIC X(<maxldb>-50); VALUE SPACES.
    02  Z-VALUE-CUST-NB-KEY; REDEFINES Z-VALUE.
        03  CUST-NB; PIC 9(10).
        03  FILLER; PIC X(<maxldb>-10); VALUE SPACES.
```

***
Those redefines clauses can be generated only with the information accessed and stored by the semantic analyser. Those informations are:
- the <maxldb>;
- the different database records type and their decomposition;
- the different access keys and their decomposition.

Let's continue the generation principles of the parameters for the interface with the GAM.
***

```
01  Z-PATHS.
    02  PATHLIST.
        03  PATHCODE; PIC 9(4); USAGE COMP; OCCURS 8 TIMES.
    02  CURLIST.
        03  CURORIGIN; PIC S9(10); USAGE COMP; OCCURS 8 TIMES.

* CODES OF THE ACTIONS OFFERED BY THE GAM
*-----------------------------------------

01  OPEN_DB; PIC 99; USAGE COMP; VALUE 01.
01  CLOSE-DB; PIC 99; USAGE COMP; VALUE 02.
01  SEQ-ACCESS; PIC 99; USAGE COMP; VALUE 03.
01  KEY-ACCESS; PIC 99; USAGE COMP; VALUE 04.
01  SEQ-AC-W-PATH; PIC 99; USAGE COMP; VALUE 05.
01  KEY-AC-W-PATH; PIC 99; USAGE COMP; VALUE 06.
01  REF-ACCESS; PIC 99; USAGE COMP; VALUE 07.
01  CREATE-REC; PIC 99; USAGE COMP; VALUE 08.
01  DELETE-REC; PIC 99; USAGE COMP; VALUE 09.
01  MODIFY-ITEMS; PIC 99; USAGE COMP; VALUE 10.
01  MODIFY-IKO; PIC 99; USAGE COMP; VALUE 11.
01  ATTACH-REC; PIC 99; USAGE COMP; VALUE 12.
01  DETACH-REC; PIC 99; USAGE COMP; VALUE 12.
01  TRANSFER-REC; PIC 99; USAGE COMP; VALUE 12.
01  REF-COMP; PIC 99; USAGE COMP; VALUE 20.
01  REF-NULL; PIC 99; USAGE COMP; VALUE 21.
01  REF-ASSIGN; PIC 99; USAGE COMP; VALUE 22.

* CODES OF THE RECORD TYPES
*--------------------------
```

***
Their generation is possible with informations found and stored by the semantic analyser. For each database record type found in the symbol table, generate:
***

```
01  <record-name>-CODE; PIC 99; USAGE COMP;
    VALUE <corresponding code used in the db>.

* CODES OF THE ACCESS PATH TYPES
*------------------------------
```

***
Their generation is possible with informations found and stored by the semantic analyser. For each access path type found in the symbol table, generate:
***

```
01  <access-path-name>; PIC 99; USAGE COMP;
    VALUE <corresponding code used in the db>.

* CODES OF THE KEYS.
*------------------
```

***
Their generation is possible with informations found and stored by the semantic analyser. For each access key found in the symbol table, generate:
***

```
01  <key-name>-CODE; PIC 99; USAGE COMP;
    VALUE <corresponding code used in the db>.

* CURRENT INSTANCES OF THE RECORD TYPES
*-------------------------------------
```

***
The generation of the current instances is possible with informations found and stored by the semantic analyser. For each variable used as reference to a database record found in the symbol table, generate:
***

```
01  CUR-<current-name>; PIC S9(10); USAGE COMP.
```

***
Those were the generation principles for the list of parameters for the interface with the general access module.

3.3. Generation principles for the variables declared in the ADL program.

Now, let's take a look at the generation principles for the variables declared in an ADL program.

The generation principles will be given for each data type of variable, either a variable is declared with a simple or structured data type, either it is declared with a non standard data type in the ADL program.

A) SIMPLE DATA TYPES.

If a variable has a simple data type or has a non standard simple data type, then that type can be only one out of five possibilities (integer, real, boolean, numeric, string). The generation principles will be given for each data type with an ADL example.

i) integer.

ADL example.

```
var A:integer
```

or

```
type INT=integer
var A:INT
```

Cobol generation:

```
01  A; PICTURE IS S9(10); USAGE COMP.
```

ii) real.

ADL example.

```
var R:real
```

or

```
type T_REAL=real
var R:T_REAL
```

Cobol generation:

```
01  R; PICTURE IS S9(8)V9(8); USAGE COMP.
```

iii) numeric.

ADL examples.

```
var NUM1:numeric(1,2);
    NUM2:numeric(5,0)

or

type  T_NUM1=numeric(1,2);
      T_NUM2=numeric(5,0)
var   NUM1:T_NUM1;
      NUM2:T_NUM2
```

Cobol generation:

```
01  NUM1; PICTURE IS S9(1)V9(2); USAGE COMP.
01  NUM2; PICTURE IS S9(5); USAGE COMP.
```

iv) boolean.

ADL example.

```
var PRESENT:boolean

or

type BOOL=boolean
var  PRESENT:BOOL
```

Cobol generation:

```
01  PRESENT; PICTURE IS A.
```

v) string.

ADL example.

```
var STR:string(20)

or

type STR20=string(20)
var STR:STR20
```

Cobol generation:

```
01  STR; PICTURE IS X(20).
```

B) STRUCTURE DATA TYPES.

If a variable has a structured data type or has a non standard structured data type, then that type can be one out of five possibilities (group, array, items_of, ref of name, ref of RECORD). The generation principles corresponding to ADL examples will be given for each structured data type.

i) group.

ADL examples.

```
var GR_1:group
      FIELD_1:integer;
      FIELD_2:group
         GR2_F1:real;
         GR2_F2:boolean
      end;
      FIELD_3:integer
    end

or

type TGR_1=group
       FIELD_1:integer;
       FIELD_2:TGR_2;
       FIELD_3:integer
     end;
     TGR_2=group
       GR2_F1:real;
       GR2_F2:boolean
     end
var GR_1:TGR_1
```

Cobol generation:

```
01  GR-1.
    02  FIELD-1; PICTURE IS S9(10); USAGE COMP.
    02  FIELD-2.
        03  GR2-F1; PICTURE IS S9(8)V9(8); USAGE COMP.
        03  GR2-F2; PICTURE IS A.
    02  FIELD-3; PICTURE IS S9(10); USAGE COMP.
```

ii) array.

Cobol admits one, two and three dimensional arrays. Here are examples of arrays of each kind.

ADL example for a one dimensional array.

```
var TAB1 : array [10] of integer

or

type T_TAB1 = array [10] of integer
var TAB1 : T_TAB1
```

Cobol generation:

```
01  TAB1.
    02  ELEMENT; OCCURS 10 TIMES; PICTURE IS S9(10);
        USAGE COMP.
```

ADL example for a two dimensional array.

```
var TAB2 : array [10,15] of real

or

type T_TAB2 = array [10,15] of real
var TAB2 : T_TAB2
```

Cobol generation:

```
01  TAB2.
    02  ELROW1; OCCURS 10 TIMES.
        03  ELROW2; OCCURS 15 TIMES; PICTURE IS S9(8)V9(8);
            USAGE COMP.
```

ADL example for a three dimensional array.

```
type GR_1=group
       F1:real;
       F2:integer
     end
var TAB3 : ARRAY [5,10,15] of GR_1
```

Cobol generation:

```
01  TAB3.
    02  ELROW1; OCCURS 5 TIMES.
        03  ELROW2; OCCURS 10 TIMES.
            04  ELROW3; OCCURS 15 TIMES.
                05  GR-1.
                    06  F1; PICTURE IS S9(8)V9(8); USAGE COMP.
                    06  F2; PICTURE IS S9(10); USAGE COMP.
```

iii) items_of

ADL example.

```
var  CUS : items_of CUSTOMER

or

type T_CUS = items_of CUSTOMER
var CUS : T_CUS
```

Cobol generation:

The Cobol generation is based on the information found by the semantic analyser and stored in the tables. Those informations are the items of the database record and their types which are implied in the current declaration (here CUSTOMER). Let's suppose the items of CUSTOMER are the name, the address and the number. The Cobol generation will be:

```
01  CUS.
    02  NAME; PICTURE IS X(20).
    02  ADDRESS; PICTURE IS X(30).
    02  NUMBER; PICTURE IS S9(3).
```

iv) ref of name.

ADL example.

```
var C1:ref of CUSTOMER
```

or

```
type R_C1 = ref of CUSTOMER
var C1 : R_C1
```

Cobol generation:

The Cobol generation is almost the same as for the "items_of" data type. Variables declared as "ref of" are structures with four different fields:

- the current of the database record stored in that structure;
- its type;
- its file;
- the items of that record.

So, the declaration of those variables implies also the search of the items of the database record and their types as in the "items_of" declaration.

So, the generation is the following:

```
01  C1.
    02  CUR; PICTURE IS S9(10); USAGE COMP.
    02  TYPE; PICTURE IS X(10).
    02  FILE; PICTURE IS X(10).
    02  ITEMS.
        03  NAME; PICTURE IS X(20).
        03  ADDRESS; PICTURE IS X(30).
        03  NUMBER; PICTURE IS S9(3).
```

v) ref of RECORD.

ADL example.

```
var REC : ref of RECORD
```

or

```
type REF_REC = ref of RECORD
var REC : REF_REC
```

Cobol generation:

A variable declared as a "ref of RECORD" is a variable that is used as a reference to a set of database records independently of their type. Its decomposition is the same as the previously discussed reference variable. However, as the records of the set it refers to are of multiple types, the structure of the data items area is variable. A redefines clause is necessary at the "item" level.
So, the generation is the following. Let's suppose we have two database record types: EMPLOYEE and WORKER.

```
01  REC.
    02  CUR; PICTURE IS S9(10); USAGE COMP.
    02  TYPE; PICTURE IS X(10).
    02  FILE; PICTURE IS X(10).
    02  ITEMS-EMPLOYEE.
        03  NAME; PICTURE IS X(20).
        03  DEPT; PICTURE IS X(10).
        03  FILLER; PICTURE X(<maxldb>-30); VALUE SPACES.
    02  ITEMS-WORKER; REDEFINES ITEMS-EMPLOYEE.
        03  NAME; PICTURE IS X(20).
        03  USINE; PICTURE IS X(15).
        03  FILLER; PICTURE X(<maxldb>-35); VALUE SPACES.
```

Let's notice, that an ambiguity problem occurs when two records have the same name for a data item. Indeed, a Cobol qualification problem will appear. For example, if we meet the following variable in a program: (REC).NAME, it is impossible to know to which database record that named item belongs to. So, the programmer has to be careful when he gives names to data items.

3.4. Remarks.

Those were the generation principles for the DATA DIVISION. Furthermore, let's say that it was impossible to present the generation of all possible combinaisons of declaration concerning the mixing of ADL data types together.

Anyway, one consideration can be here added. In ADL, a variable declaration can declare more than one identifier in the same declaration. Cobol does not allow that; so when such ADL declaration occurs, the Cobol program has to contain X declarations of variables of the same type (X is the number of identifiers declared in the ADL declaration statement).

Example.

```
var A,B:integer

Cobol generation:

01  A; PICTURE IS S9(10).
01  B; PICTURE IS S9(10).
```

4. GENERATION PRINCIPLES FOR THE PRODEDURE DIVISION.

4.1. Introduction.

This section presents the generation principles for the PROCEDURE DIVISION. First, we give the generation principles for all kinds of variables, all kinds of arithmetic expressions and general expressions. Then, the general structure of this DIVISION is given and the choice of the control structure will be explained. And finally, the generation principles for all ADL statements will be given.

4.2. Generation principles for the variables, the arithmetic expressions and the general expressions.

As each ADL statement contains in some way variables, arithmetic expressions and general expressions, let's first present their generation principles.

4.2.1. The variables.

For each kind of ADL variable, the corresponding generation principles will be given for an example.

4.2.1.1. Non hierarchical variables.

- non subscripted variables.

  ADL variable: ORDER ——> Cobol generation: ORDER

- subscripted variables.

  ADL variable: TAB[I] ——> Cobol generation: TAB(I)

4.2.1.2. Hierarchical variables.

ADL variable: AD.STR ——> Cobol generation: STR OF AD

4.2.1.3. Classical var items, var items file and var items type.

ADL variables.		Cobol generation.
(CUS).NAME	——>	NAME OF CUS
(CUS).FILE	——>	FILE OF CUS
(CUS).TYPE	——>	TYPE OF CUS

4.2.2. The arithmetic expressions.

The arithmetic expression contains a certain number of primarys linked together by arithmetic operators. The generation principles for an arithmetic expression is the following: generate a primary, the arithmetic operator and the next primary, if any and so on.

ADL examples.		Cobol generation.
A+B	——>	A + B
-(A*B)	——>	- (A * B)
(A/B)^2	——>	(A / B) ** 2
(A*5)+(B/0.1)	——>	(A * 5) + (B / 0.1)

Remark.

If the call statement appears in an arithmetic expression (or in a general expression), its generation principle is the following. Before the statement in which this arithmetic expression (or general expression) appears, a Cobol call must be generated with the parameters of the ADL call as arguments of that Cobol call. The last argument will contain the result of the Cobol call and this argument will be generated in the arithmetic expression as substitute of the call statement. An example of that particular case will be given in the generation principles of the assignment statement.

4.2.3. The general expressions.

The general expression contains a certain number of general primarys linked together by logical operators. The generation principles for a general expression is, thus, the following: generate a general primary and then, if any, the binary (AND, OR) or unary (NOT) operator and the following general primary, and so on. If a general primary is a test expression, then generate the first arithmetic expression, the test operator and the second arithmetic expression.

ADL examples.		Cobol generation.
(A<B)	——>	(A IS < B)
A<>B	——>	A IS NOT = B
(A>=B) and (C<=D)	——>	(A IS NOT = B) AND (C IS NOT > D)
(A=B) or not (C>D)	——>	(A IS = B) OR NOT (C IS > D)

Remark.

If the test of the general expression is an equality test of two reference variables, then that test is done thanks to a call to a function of the GAM. This call is performed before the statement where the general expression appears.

Example.

```
if (REF1=REF2) then ...
```

Cobol generation.

```
MOVE ZEROS TO Z-CODES.
MOVE SPACES TO Z-VALUES, Z-PATHS.
MOVE REF-COMP TO OPCODE.
MOVE CUR-REF1 TO CURORIGIN(1).
MOVE CUR-REF2 TO CURORIGIN(2).
CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
IF ERRCODE IS = err_eq1;
    MOVE "T" TO REF-EQ;
    ELSE IF ERRCODE IS = err_dif;
         MOVE "F" TO REF-EQ;
         ELSE GO TO TR-ERROR.
IF (REF-EQ IS = "T");
...
```

4.3. The general structure of the PROCEDURE DIVISION.

4.3.1. The control structure chosen: GO TO vs PERFORM.

The two possible control structures within the PROCEDURE DIVISION are the GO TO and the PERFORM control structures. The main hindrance to the use of PERFORMs is the ADL next name statement. Indeed, when a next name statement is encountered, it means that it forces the abortion of the current loop execution. If we had used the perform for the generation, a recursive PERFORM would be necessary. This is not accepted in Cobol. So, as a GO TO is necessary for the generation of the next statement, we decided that the GO TO control structure should be used instead of the PERFORM control structure for the generation of all ADL statements. All the same, two PERFORMs are generated in the Cobol program but in the general structure of the PROCEDURE DIVISION, as it will be seen in the next section.

4.3.2. General structure of the PROCEDURE DIVISION.

Here is the general structure of the PROCEDURE DIVISION. This structure will always be generated even if the statement part of the ADL program is empty.

Cobol generation.

```
PROCEDURE DIVISION.
*******************
MAIN-PROGRAM.
    PERFORM OVERTURE-DB.
    GO TO MAIN-TRT.
ENDST.
    PERFORM CLOTURE-DB.
    STOP RUN.
OVERTURE-DB.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE OPEN-DB TO OPCODE.
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE = 2; MOVE 1 TO DBALOPEN;
        ELSE IF ERRCODE NOT = 0; GO TO TR-ERROR.

***
The initialization of all variables used as reference is necessary. It is
done just after the opening of the database by calling a function of the
GAM (REF-NULL) for each variable that is used as a reference. They are
found in the symbol table. For each reference variable, generate:
***

    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE REF-NULL TO OPCODE.
    MOVE CUR-‹reference variable› TO CURORIGIN(1).
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE NOT = 0; GO TO TR-ERROR.

***
Then the generation continues with the cloture-db paragraph.
***

CLOTURE-DB.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE CLOSE-DB TO OPCODE.
    IF DBALOPEN = 0;
        CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS;
        IF ERRCODE NOT = 0; GO TO TR-ERROR.
MAIN-TRT.
    GO TO ENDST.
```

```
TR-ERROR.
    DISPLAY 'ERROR IN DATABASE'.
    DISPLAY DBSTAT.
    STOP RUN.
```

Remarks.

1) The 2 PERFORMs actions of the MAIN-PROGRAM of the Cobol program generated are executed once. The OVERTURE-DB paragraph performs the opening of the database and the initialization of all the currents used in the program by using a function offered by the GAM: REF-NULL, that either cancel a reference either assign the null reference. The CLOTURE-DB paragraph performs the closing of the database, if it was closed before the opening. The MAIN-TRT paragraph performs all the actions that are present in the statement part of the ADL program. Their generation principles will be given in the next section. Finally, the TR-ERROR paragraph is used when a fatal database error occurs (fatal meaning that the execution of the program cannot continue without spoiling the database). TR-ERROR informs the user that an error occurs, it also displays the error code and the operation code and stops the execution of the program.

2) Each time a call the DB (the access module) is made the following steps are performed. Those steps are presents in all calls so far generated:

- initialization to zero of all areas that are used as interface with the GAM;
- assignation of the parameters used in the current call;
- call to DB (the access module program);
- if an error occurs then perform the appropriate process;
- if the operation is a database access, it is necessary to stored the items and the reference of the accessed record.

3) It is obvious that DBALOPEN and REF-EQ are to be added in the WORKING-STORAGE SECTION:

```
01  DBALOPEN; PIC 9; USAGE COMP.
01  REF-EQ; PIC A.
```

4.4. Generation principles for the ADL statements.

4.4.1. General structure of the MAIN-TRT paragraph.

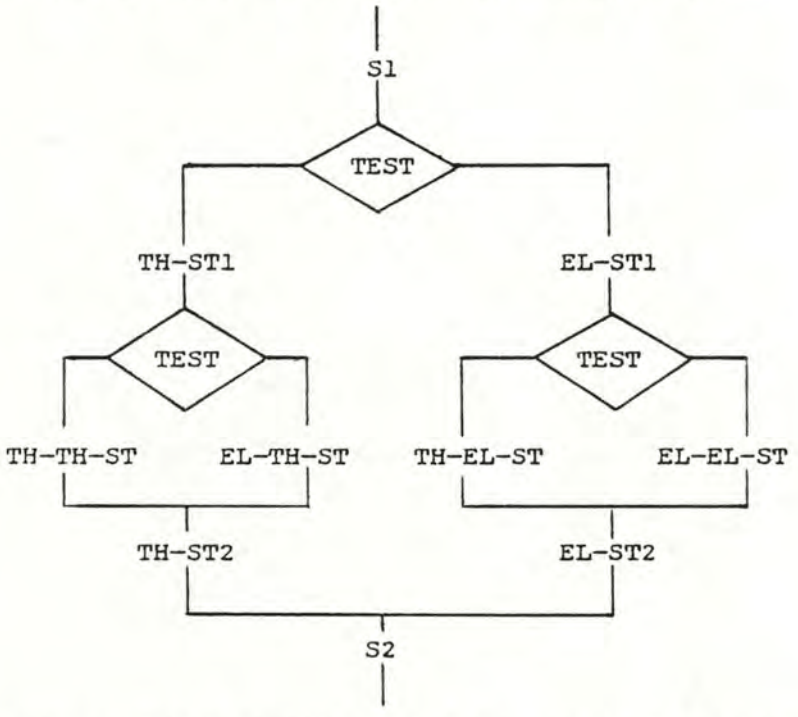
The choice that was made for the Cobol generation of the ADL statement part is the following. Each ADL statement is generated according to its generation principles. These principles will be presented in the next sections.

If the ADL statement is an ADL control structure statement (if, while, for) then the sequences of statements controled by this statement are not immediately generated. Each of those sequences is generated when each ADL statement present on, what we can call, the first level of statements, has

been generated. The first level in an ADL program is the set of statements that are not contained in any control structure. The second level is the set of sequences of statements contained in the control statement of the first level, and so on.

In other words, we can summarize this process by saying:
For each statement of the sequence of statements on the current level, if the ADL statement is not a control structure statement then generate this statement. Otherwise, generate the control structure but not the sequence of statements controled by this control structure statement.
When the whole sequence of statements of the current level has been generated, then generate the sequences of statements controled by the control structure statements of that level; and so forth, recursively.

This structure was mainly chosen because of the possibility of overlapping if statement. Indeed, if a sequence of statements controled by the if statement contains another if statement, then the following structure is impossible in the generated Cobol text.



So, to avoid this problem, the sequences of statements controled by the if control structure statement are generated later. As this principle was chosen for the if control structure statement, it was adopted for all control structure statements.

4.4.2. Generation principles: the assignment expression.

In our effort to give a complete list of all possible types of assignment statement, we group the examples in three blocks:

- the first gives the different forms of arithmetic expressions and the different kinds of primary, excepted the variable used as a reference to a record and the particular case when the primary is a db object

set and nullref;
- the second gives the forms of arithmetic expressions and primary which are the exceptions of the first group;
- the third group states examples of logical expressions: general expressions.

4.4.2.1. Arithmetic expression and primary.

ADL example. Cobol generation.

```
A:=(B+C-D)*E;        ---->    COMPUTE A = (B + C - D) * E.
A:=B;                ---->    MOVE B TO A.
A[5]:=5;             ---->    MOVE 5 TO A(5).
ADR:='XXX';          ---->    MOVE "XXX" TO ADR.
PRESENT:=true;       ---->    MOVE "T" TO PRESENT.
A:= 0.5e3 + 0.8;     ---->    COMPUTE A = (0.5 ** + 3) + 0.8.
A:=SIN!(B,SINB)      ---->    CALL "SIN" USING B, SINB.
                              MOVE SINB TO A.
```

4.4.2.2. Particular cases of the primary.

A. Db object set.

ADL example.

```
CUS:=CUSTOMER( :CUSNB=5 )
```

Cobol generation.

```
MOVE ZEROS TO Z-CODES.
MOVE SPACES TO Z-VALUES, Z-PATHS.
MOVE KEY-ACCESS TO OPCODE.
MOVE 1 TO GETCODE.
MOVE CUSTOMER-CODE TO RECTCODE.
MOVE CUR-CUS TO RECREF.
MOVE CUSNB-CODE TO KEYCODE.
MOVE 0 TO OPERATOR.
MOVE 5 TO Z-VALUE-CUSNB-KEY.
CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
IF ERRCODE NOT = 0;
   IF ERRCODE NOT = 1; GO TO TR-ERROR.
MOVE RECREF TO CUR-CUS.
MOVE CORRESPONDING Z-VALUE-CUSTOMER TO CUS.
```

That particular case of the assignment statement corresponds to a single access key to a database record type.

B. Nullref and variable used as reference to a record.

ADL examples.

```
CUR_X:=( );
CUR_X:=CUR_Y;
```

Cobol generation.

```
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE REF-NULL TO OPCODE.
    MOVE CUR-X TO CURORIGIN(1).
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE NOT = 0; GO TO TR-ERROR.

    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE REF-ASSIGN TO OPCODE.
    MOVE CUR-X TO CURORIGIN(1).
    MOVE CUR-Y TO CURORIGIN(2).
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE NOT = 0; GO TO TR-ERROR.
```

These particular cases of assignation correspond to the cancellation of a reference and the assignation of a reference to another.

4.4.2.3. General expression.

ADL example.

```
PRESENT := (A>B[I]) and (A<B[J]);
```

Cobol generation.

```
    IF (A IS > B(I)) AND (A IS < B(J));
         MOVE "T" TO PRESENT;
    ELSE MOVE "F" TO PRESENT.
```

4.4.3. Generation principles: the while statement.

For each control structure (while, if ,for), we generate two sequences of Cobol statements at two different places. These two sequences are the statements concerning the control structure and then at the following level the statements controled by this structure. The separation of the two parts is indicated in the examples with "////".

ADL example.

```
while A <> B[I] do
      I:=I+1
endwhile;
```

Cobol generation.

```
WH-i.
    IF A NOT = B(I);
    GO TO WHST-i.
////
WHST-i.
    COMPUTE I = I + 1.
    GO TO WH-i.
```

Where i is the counter of the number of while statements.

4.4.4. Generation principles: the if statement.

This section presents the generation principles for both kinds of if: if_then statement and if_then_else statement.

A. If_then statement.

ADL example.

```
if QTY=0
   then PRINT!(MESS_ERR);
endif
```

Cobol generation.

```
    IF QTY IS = 0;
        GO TO TH-ST-i.
END-IF-i.
////
TH-ST-i.
    CALL "PRINT" USING MESS-ERR.
    GO TO END-IF-i.
```

Where i is the counter of the number of if statements.

B. If_then_else statement.

ADL example.

```
if QTY = 0
   then PRINT!(MESS_ERR)
   else QTY:=QTY-(L).QTY
endif;
```

Cobol generation.

```
    IF QTY IS = 0;
         GO TO TH-ST-i;
    ELSE GO TO EL-ST-i.
END-IF-i.
////
TH-ST-i.
    CALL "PRINT" USING MESS-ERR.
    GO TO END-IF-i.
EL-ST-i.
    COMPUTE QTY = QTY - QTY OF L.
    GO TO END-IF -i.
```

Where i is the counter of number of if statements.

4.4.5. Generation principles: the for statement.

In ADL, the for statement can be either a for with a range as collection expression either a for with a db object set as collection expression. This section presents the generation principles for both kind of for statements.

4.4.5.1. Generation principles for the for range.

ADL example.

```
for A:=1..10 do
    TAB[A] := A^2
endfor
```

Cobol generation.

```
FOR-R-i.
    IF A NOT > 10;
    GO TO FOR-R-ST-i.
////
FOR-R-ST-i.
    COMPUTE TAB(A) = A ** 2.
    COMPUTE A = A + 1.
    GO TO FOR-R-i.
```

Where i is the counter of the number of for range statements.

4.4.5.2. Generation principles for the for db object set statement.

The generation of the for statement with a db object set as collection expression is a little bit more complex. Let's recall that this for statement always represents an access to one, several or all instances of a given record type in the database. This kind of for statement, also called for_db, is generated as a loop of accesses to database records using the functions offered by the GAM. There are four possible types of accesses. The generation principles will be given for each type of access with an example. These types of accesses are:

- the sequential access to instances of a database record type;
- the access to one or more instances of a record type based on a access key;
- the sequential access to the targets of an access path;
- the access to the targets of an access path based on a key.

A. Sequential access to instances of a record type.

ADL example.

```
for CUS:=CUSTOMER( ) do
       ‹trt›
endfor;
```

Cobol generation.

```
    GO TO TR-FORDB-i.
END-FORDB-i.
////
TR-FORDB-i.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE SEQ-ACCESS TO OPCODE.
    MOVE 1 TO GETCODE.
    MOVE CUSTOMER-CODE TO RECTCODE.
    MOVE CUR-CUS TO RECREF.
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE NOT = 0;
        IF ERRCODE NOT = 1; GO TO TR-ERROR;
        ELSE GO TO END-TRFORDB-i.
    MOVE RECREF TO CUR-CUS.
    MOVE CORRESPONDING Z-VALUE-CUSTOMER TO CUS.
    GO TO TR-FORDBST-i.
END-TRFORDBST-i.
    GO TO TR-FORDB-i.
END-TRFORDB-i.
    GO TO END-FORDB-i.
////
TR-FORDBST-i.
    ‹trt›
    GO TO END-TRFORDBST-i.
```

Where i is the counter of the number of for_db statement.

B. Access to instances of a record type based on a key.

ADL example.

```
for CUS:=CUSTOMER(( :CUS_NAME='DUPONT' )and( :CUS_ADR='NAMUR' )) do
        <trt>
endfor
```

Cobol generation.

```
    GO TO TR-FORDB-i.
END-FORDB-i.
////
TR-FORDB-i.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE KEY-ACCESS TO OPCODE.
    MOVE 1 TO GETCODE.
    MOVE CUSTOMER-CODE TO RECTCODE.
    MOVE CUR-CUS TO RECREF.
    MOVE "DUPONT" TO CUS-NAME OF Z-VALUE-CUS-NAME-DEC-KEY.
    MOVE "NAMUR" TO CUS-ADR OF Z-VALUE-CUS-NAME-DEC-KEY.
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE NOT = 0;
        IF ERRCODE NOT = 1; GO TO TR-ERROR;
        ELSE GO TO END-TRFORDB-i.
    MOVE RECREF TO CUR-CUS.
    MOVE CORRESPONDING Z-VALUE-CUSTOMER TO CUS.
    GO TO TR-FORDBST-i.
END-TRFORDBST-i.
    GO TO TR-FORDB-i.
END-TRFORDB-i.
    GO TO END-FORDB-i.
////
TR-FORDBST-i.
    <trt>
    GO TO END-TRFORDBST-i.
```

Where i is the counter of the number of for_db statement.

C. Sequential access to the targets of an access path.

ADL example.

```
for ORD:=ORDER(CO:CUS) do
      <trt>
endfor;
```

Cobol generation

```
    GO TO TR-FORDB-i.
END-FORDB-i.
////
TR-FORDB-i.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE SEQ-AC-W-PATH TO OPCODE.
    MOVE 1 TO GETCODE.
    MOVE ORDER-CODE TO RECTCODE.
    MOVE CUR-ORD TO RECREF.
    MOVE CO TO PATHCODE(1).
    MOVE CUR-CUS TO CURORIGIN(1).
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE NOT = 0;
        IF ERRCODE NOT = 1; GO TO TR-ERROR;
        ELSE GO TO END-TRFORDB-i.
    MOVE RECREF TO CUR-ORD.
    MOVE CORRESPONDING Z-VALUE-ORDER TO ORD.
    GO TO TR-FORDBST-i.
END-TRFORDBST-i.
    GO TO TR-FORDB-i.
END-TRFORDB-i.
    GO TO END-FORDB-i.
////
TR-FORDBST-i.
    <trt>
    GO TO END-TRFORDBST-i.
```

Where i is the counter of the number of for_db statement.

D. Access to the targets of an access path based on a key.

ADL example.

```
for ORD:=ORDER((CO:CUS)and(:DATE_ORD=DATE_X)) do
      ‹trt›
endfor;
```

Cobol generation.

```
    GO TO TR-FORDB-i.
END-FORDB-i.
////
TR-FORDB-i.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE KEY-AC-W-PATH TO OPCODE.
    MOVE 1 TO GETCODE.
    MOVE ORDER-CODE TO RECTCODE.
    MOVE CUR-ORD TO RECREF.
    MOVE DATE-ORD-CODE TO KEYCODE.
    MOVE DATE-X TO Z-VALUE-DATE-ORD.
    MOVE 0 TO OPERATOR.
    MOVE CO TO PATHCODE(1).
    MOVE CUR-CUS TO CURORIGIN(1).
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE NOT = 0;
        IF ERRCODE NOT = 1; GO TO TR-ERROR;
        ELSE GO TO END-TRFORDB-i.
    MOVE RECREF TO CUR-ORD.
    MOVE CORRESPONDING Z-VALUE-ORDER TO ORD.
    GO TO TR-FORDBST-i.
END-TRFORDBST-i.
    GO TO TR-FORDB-i.
END-TRFORDB-i.
    GO TO END-FORDB-i.
////
TR-FORDBST-i.
    ‹trt›
    GO TO END-TRFORDBST-i.
```

Where i is counter of the number of for_db statement.

4.4.6. Generation principles: the exit and the next statements.

The generation principles for the next statement and the generation principles for the exit statement are quite similar. The main difference is the statement where the execution must continue after the exit or the next statement. The exit statement presence implies that the execution continues at the END-TRFORDB-i paragraph while the next statement presence implies that the execution continues at the END-TRFORDBST-i paragraph (where i corresponds to the number of the for statement whose control variable is the name of the exit or of the next statement, if any; otherwise, i the number of the current generated for statement).

ADL example.

```
for CUS:=CUSTOMER( ) do
   for ORD:=ORDER(CC:CUS) do
      if (ORD).DATE=310885
         then PRINT!((CUS).NAME)
         else next CUS
              [exit]
      endif
   endfor
endfor
```

Cobol generation.

```
    GO TO TR-FORDB-1.
END-FORDB-1.
////
TR-FORDB-1.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE SEQ-ACCESS TO OPCODE.
    MOVE 1 TO GETCODE.
    MOVE CUSTOMER-CODE TO RECTCODE.
    MOVE CUR-CUS TO RECREF.
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE NOT = 0;
        IF ERRCODE NOT = 1; GO TO TR-ERROR;
        ELSE GO TO END-TRFORDB-1.
    MOVE RECREF TO CUR-CUS.
    MOVE CORRESPONDING Z-VALUE-CUSTOMER TO CUS.
    GO TO TR-FORDBST-1.
END-TRFORDBST-1.
    GO TO TR-FORDB-1.
END-TRFORDB-1.
    GO TO END-FORDB-1.
////
TR-FORDBST-1.
    GO TO FORDB-2.
END-FORDB-2.
```

```
    GO TO END-TRFORDBST-1.
////
TR-FORDB-2.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE SEQ-AC-W-PATH TO OPCODE.
    MOVE 1 TO GETCODE.
    MOVE ORDER-CODE TO RECTCODE.
    MOVE CUR-ORD TO RECREF.
    MOVE CO TO PATHCODE(1).
    MOVE CUR-CUS TO CURORIGIN(1).
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE NOT = 0;
        IF ERRCODE NOT = 1; GO TO TR-ERROR;
        ELSE GO TO END-TRFORDB-2.
    MOVE RECREF TO CUR-ORD.
    MOVE CORRESPONDING Z-VALUE-ORDER TO ORD.
    GO TO TR-FORDBST-2.
END-TRFORDBST-2.
    GO TO TR-FORDB-2.
END-TRFORDB-2.
    GO TO END-FORDB-2.
TR-FORDBST-2.
    IF DATE OF ORD IS = 310885;
         GO TO TH-ST-1;
    ELSE GO TO EL-ST-1.
END-IF-1.
    GO TO END-TRFORDBST-2.
TH-ST-1.
    CALL "PRINT" USING NAME OF CUS.
    GO TO END-IF-1.
EL-ST-1.
    GO TO END-TRFORDBST-1.
  [ GO TO END-TRFORDB-2. ]
    GO TO END-IF -1.
```

4.4.7. Generation principles: the call statement.

The call statement is generated by using the CALL Cobol statement.

ADL examples.		Cobol generation.
FUNC!	——>	CALL "FUNC".
FUNC!(ARG1,ARG2)	——>	CALL "FUNC" USING ARG1, ARG2.

4.4.8. Generation principles: the return statement.

The ADL return statement is generated in Cobol, by only one clause either it has parameters or either it has no parameter. This clause is the EXIT PROGRAM clause that can be the only statement of its paragraph.

ADL examples.

```
return;
return(ARG1);
```

Cobol generation.

```
    GO TO EXPROG.
////
EXPROG.
    EXIT PROGRAM.
```

4.4.9. Database modification statements.

ADL offers three kinds of database modification statements: create, delete and modify. We will give generation principles for each of these statements.

4.4.9.1. Generation principles: the create statement.

ADL example.

```
create ORD:=ORDER((CO:CUS) and (:DATE_ORD=DATE_TODAY) and
                  (:NUM_ORD=LASTNB_ORD +1);
```

Cobol generation.

```
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE CREATE-REC TO OPCODE.
    MOVE ORDER-CODE TO RECTCODE.
    MOVE CO TO PATHCODE(1).
    MOVE CUR-CUS TO CURORIGIN(1).
    MOVE DATE-TODAY TO DATE-ORD OF ORD.
    COMPUTE NUM-ORD OF ORD = LASTNB-ORD + 1.
    MOVE CORRESPONDING ORD TO Z-VALUE-ORDER.
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE NOT = 0; GO TO TR-ERROR.
    MOVE RECREF TO CUR-ORD.
```

4.4.9.2. Generation principles: the delete statement.

ADL example.

```
delete ORD
```

Cobol generation.

```
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE DELETE-REC TO OPCODE.
    MOVE ORDER-CODE TO RECTCODE.
    MOVE CUR-ORD TO RECREF.
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE NOT = 0; GO TO TR-ERROR.
    MOVE RECREF TO CUR-ORD.
```

4.4.9.3. Generation principles: the modify statement.

The ADL modify statement offers ways to modify data items of a database record but it also allows to attach and detach records to others via access paths. This section will present the generation principles for the modification of data items (or ikos), for the detach operation and the detach operation.

A. Modify items (or ikos) statement.

The GAM offers two modify functions: modify-items and modify-ikos. When an ADL modify statement is encountered, the Cobol program will contain back to back the call to the modify-items function and the call to the modify-ikos function. Each of them will modify the data items or the ikos it can modify (data items for the modify-items function and the ikos for the modify-ikos function).

ADL example.

```
modify CUS(( :ADR_CUST='XXX' )and( :CITY_CUST='YYY' ));
```

Cobol generation.

```
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE MODIFY-ITEMS TO OPCODE.
    MOVE CUSTOMER-CODE TO RECTCODE.
    MOVE CUR-CUS TO RECREF.
    MOVE "XXX" TO ADR-CUST OF Z-VALUE-CUSTOMER.
    MOVE "YYY" TO CITY-CUST OF Z-VALUE-CUSTOMER.
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
```

```
IF ERRCODE NOT = 0; GO TO TR-ERROR.
MOVE ZEROS TO Z-CODES.
MOVE SPACES TO Z-VALUES, Z-PATHS.
MOVE MODIFY-IKOS TO OPCODE.
MOVE CUSTOMER-CODE TO RECTCODE.
MOVE CUR-CUS TO RECREF.
MOVE "XXX" TO ADR-CUST OF Z-VALUE-CUSTOMER.
MOVE "YYY" TO CITY-CUST OF Z-VALUE-CUSTOMER.
CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
IF ERRCODE NOT = 0; GO TO TR-ERROR.
```

B. The attach statement.

ADL example.

```
modify ORD(CO:CUS)
```

Cobol generation.

```
MOVE ZEROS TO Z-CODES.
MOVE SPACES TO Z-VALUES, Z-PATHS.
MOVE ATTACH-REC TO OPCODE.
MOVE CUR-ORD TO RECREF.
MOVE CO TO PATHCODE(1).
MOVE CUR-CUS TO CURORIGIN(1).
CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
IF ERRCODE NOT = 0; GO TO TR-ERROR.
```

C. The detach statement.

ADL example.

```
modify ORD(CO: 0 CUS)
```

Cobol generation.

```
MOVE ZEROS TO Z-CODES.
MOVE SPACES TO Z-VALUES, Z-PATHS.
MOVE DETACH-REC TO OPCODE.
MOVE CUR-ORD TO RECREF.
MOVE CO TO PATHCODE(1).
MOVE CUR-CUS TO CURORIGIN(1).
CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
IF ERRCODE NOT = 0; GO TO TR-ERROR.
```

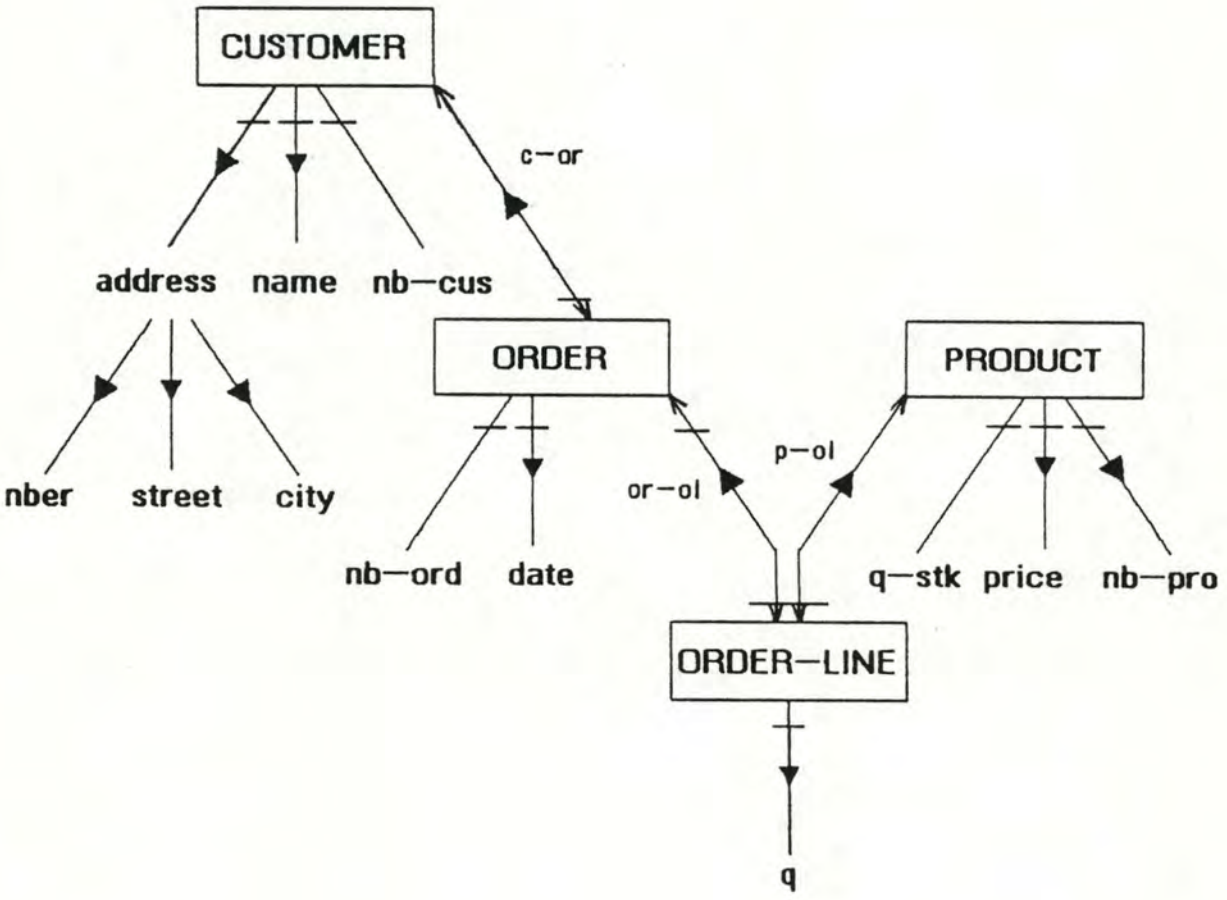
CHAPTER 8: EXAMPLE OF AN APPLICATION

1. INTRODUCTION

This chapter presents the example of an application. This includes first the access schema (necessary access schema) of the database. The database stores information about customers, their orders and the products which may be ordered. Then, we specify a request and give the corresponding effective ADL algorithm. The ADL program is given as input to the ADL/GAM Cobol compiler. The results of the execution (the decompilation of the generated parse tree plus the decompiled source text and the Cobol program) can be found in the appendix.

2. DESCRIPTION OF THE SCHEMA

2.1. NAS schema



2.2. Description

A description of the database record types, their data items and the access paths linking the record types are presented for the preceding schema.

RECORD TYPE: CUSTOMER

CUSTOMER: person or firm who has at least once ordered one of our products

DATA ITEMS:
- NB_CUS: the number identifying the customer; type: integer;
- NAME: the name of the customer; type: string;
- ADDRESS: the address of the customer; type: decomposable
  Component data items:
  * NBER: number of the house; type: integer;
  * STREET: name of the street; type: string;
  * CITY: name of the town or village; type: string;

RECORD TYPE: ORDER

ORDER: order passed by a customer containing at least one correct order line

DATA ITEMS:
- NB_ORD: number identifying the order; type: integer;
- DATE: date specifying the day the order was passed; type: integer (the two first digits give the day, the third and fourth the month and the last two the year);

RECORD TYPE: ORDER_LINE

ORDER_LINE: line of an order specifying the quantity of an ordered product

DATA ITEMS:
- Q: quantity ordered; type: integer;

RECORD TYPE: PRODUCT

PRODUCT: good produced by the firm and that can be bought by customers

DATA ITEMS
- NB_PRO: number identifying the product; type: integer;
- PRICE: unit selling price of the product; type: real;
- Q_STK: quantity of the product that is available in the stock of the firm; type: integer;

ALL ITEMS ARE MANDATORY.

ACCESS PATH TYPES

C_OR: access path linking a customer to its orders;
(one-to-many);
OR_C: inverse; access path linking an order to the customer who passed it;
(many-to-one);
OR_OL: access path linking an order and its order lines;
(one-to-many);
OL_OR: inverse; access path linking an order line to its order;
(many-to-one);
P_OL: access path linking a product to all its order lines;
(one-to-many);
OL_P: inverse; access path linking an order line to its corresponding product; (many-to-one);

3. REQUEST & ADL ALGORITHM

3.1. Request

Count the number of customers ordering the product number 5159 on August 31st 1985.

3.2. ADL algorithm

```
algorithm FUNC1NBCUST

var CNBCUST: integer;
    CUST : ref of CUSTOMER;
    ORD : ref of ORDER;
    OL: ref of ORDER_LINE;
    PROD: ref of PRODUCT
begin
CNBCUST:=0;
for CUST:=CUSTOMER( ) do
   for ORD:=ORDER(C_OR:CUST) do
      if (ORD).DATE = 310885
         then
            for OL:=ORDER_LINE(OR_OL:ORD) do
               for PROD:=PRODUCT(OL_P:OL) do
                  if (PROD).NB_PRO = 5159
                    then
                       CNBCUST:=CNBCUST+1;
                       next CUST;
                  endif;
               endfor;
            endfor
      endif
   endfor
endfor
end.
```

CONCLUSIONS

PART 1 CONCLUSIONS ON THE WORK CARRIED OUT

1. Our Work

The objective of our work was the implementation of an ADL-Cobol/GAM compiler within the development of a database design workbench. Being an automated tool which is very specific to the design methodology, the first part of the thesis was exclusively dedicated to a very brief introduction to the analysis and design methodology of Namur and to the presentation of the models supporting the realization of the software environment (the GAM and ADL). A special attention has also been paid to the conceptual schema of the database of the workbench.

The second part of the work then described the choices that have been made in the design as well as in the implementation of the compiler. As it represents one tool among others of the workbench, its design has mostly been determined by the structure of the global environment. Emphasis being put on the production of modular automated tools in order to make the decisions more independent, the three steps of the compiler (syntactical analysis, semantic analysis, code generation) have been clearly separated in their design.

2. Evaluation

A. SYNTACTICAL ANALYSIS

A syntactical analyser has been created for the effective ADL programs as well as for the still incomplete predicative (general or whole) ADL. The tools LEX and YACC used to generate the parse tree proved to be very helpful and efficient. They enabled us to implement the ADL parser in a considerably short time. However, an important thing to note is that these UNIX tools only allow the generation of LALR(1) parsers. In other words, a lookahead of one token should in all cases allow the recognition of the current syntactic structure. Grammar rules ambiguous for the parser are handled in a fixed way which may be unsatisfactory. There is no possibility of backtracking if a wrong interpretation choice has been made. So, to assure a defined and correct behaviour of the generated parser, the specified grammar has to be unambiguous. This restriction proved to be a real hindrance for the specification of the ADL language. Besides the introduction of some special symbols, we had to limit the power of the database access constructions. Thus a parser of a more general ADL has still to be implemented.

B. SEMANTIC ANALYSIS & CODE GENERATOR

The semantic analyser and the consequent code generator only concern effective ADL programs. They have been conjointly developed and so their interactions are considerable. Besides the various consistency checks, the semantic analyser provides the generation process with informations necessary to its execution. So, as the compiler is strictly limited to the generation into one programming language (Cobol), the semantic analyser also includes some functions of information collecting which may become completely meaningless in case of another target programming language. The compiler itself is completely dependent of Cobol. Moreover, as the product is a first prototype for the automatic generation of executable programs, it has of course its limitations. For instance, it hardly handles the problem of the function calls and does not tackle the question of communication between programs at all.

C. GENERATED PRODUCTS

An other interesting point is the evaluation of the generated products. The parse tree generated by the syntactic analyser forms the algorithm representation within the system. All the transformations which have to be performed on the algorithm according to the steps of the logical and physical design will be executed on this tree. So the number of nodes generated for a given program indirectly determines the performance of the tree transformation tools. The examples given in the appendix show that the complexity of the ADL grammar produces a large number of nodes, even for very short ADL programs. From a different point of view, this decomposition down to the basic elements of the language may facilitate the transformations themselves. So it remains rather difficult to give a definite judgement on this subject. Experience will be more eloquent.

PART 2: EXTENSIONS

What could have been done differently with regard to possible future developments?

1. Multiple Target Languages

The implemented code generator is strictly limited to the generation into one defined target programming language. The characteristics of the chosen language (Cobol in this case) determined the design of the compiler. Whereas the semantic analyser collects rather general informations, even if some of it may be irrelevant for the translation into some other programming language, the generator itself is totally dependent of Cobol.

Considering the whole context of the database design workbench, it may have been more interesting to design a compiler where the target-language dependency is confined to a small number of basic functions, thus making the development of a compiler with some other target programming language much easier. In concrete terms, the compiler would be based upon a series

of translating functions, one for each basic instruction of the ADL language. So, the main structure of the program will always remain the same; only these translation functions have to be rewritten for each target language.

In this context, it becomes interesting to accept for the effective ADL programs a language with only very elementary control structures. This is all the more necessary as the target programming languages may not all offer powerful algorithmic structures equivalent to the ADL structures. The translation of the complex ADL control structures would be performed by tree transformations.

A FOR structure of the form :

```
for X := RECORD_X(CX)
    <statements>
endfor
```

will be transformed into :

```
GET_FIRST_RX;
while FOUND_X do
    <statements>;
    GET_NEXT_RX
endwhile
```

where : GET_FIRST_RX gets the first record RECORD_X that satisfies the condition CX,
GET_NEXT_RX gets the next record of the sequence of records of type RECORD_X satisfying the condition CX.
These functions return a boolean value indicating if the record has been found (FOUND_X) and do also return the reference of this record.

A WHILE loop is again transformed into an IF statement :

```
LABEL1: if not <condition> then GOTO LABEL2 endif;
        <statements>
        GOTO LABEL1;
LABEL2:
```

At the same time, the power of the ADL expressions is restricted by allowing only binary arithmetic expressions and simple test expressions. These transformations may imply the introduction of auxiliary variables. This way, the set of necessary translation routines specific to one target programming language is considerably reduced and so is the dependency of the compiler.

Given this architecture, it shouldn't be too difficult to implement a multiple target generator. A parameterization of the whole module could for instance be used to determine which subset of the translating functions should be used to obtain a program text in the chosen conventional programming language. An inevitable consequence of this way of proceeding however, is that the generated program text will not use all the power of the given programming language, thus being less readable.

A second way for implementing a general generation tool of executable programs could be the design of some program which accepts as input the parse tree of the ADL program as well as a file of macro-instructions. The instructions describe the principles for translating the various structures of the ADL language into a particular target language. So, to achieve the translation into yet another programming language, only the file of the macro-instructions has to be modified. This type of architecture has been used for instance to produce a general automatic generator of access modules (see [BOCK.82]).

2. Generation of Modular Access Modules

From another point of view, it may be profitable to add some functions to the compiler like the collecting of additional informations. They will be stored in the database of the workbench in order to be generally available. These specific informations could for instance be the list of all database manipulations performed in the algorithm so that modular access interface modules can be generated. We mean by this that the semantic analyser will have to save detailed information about what primitive access and modification routines are requested for each record type of the database schema. So it is not enough to store the record type names manipulated in a given algorithm; for each record type, the operations to be executed must be listed or, vice versa for each possible database operation, any record type for which this action is requested has to be stored.

If this extra information is entered into the database of the workbench, it should be possible to generate modular access modules, that's to say access modules associated to one special algorithm and containing as code only what is strictly necessary to the execution of this application.

It goes without saying that we do not claim presenting here all the extensions which may be possible. The implemented compiler only represents a first step in the realization of an automated translation tool of the workbench.

BIBLIOGRAPHY

[AHO.77] A.V. AHO & J.D. ULLMAN
"Principles of Compiler Design."
Addison-Wesley, 1977

[BAUE.76] BAUER, DE REMER, ERSHOV, GRIES, GRIFFITHS, HILL, HORNING, KOSTER, McKEEMAN, POOLF, WAITE
"Compiler Construction- An Advanced Course."
F.L. Bauer & J. Eickel, 2nd edition, Springer Verlag

[BOCK.82] P. BOCK & M. DELVAL
"Generation automatique d'interfaces d'acces a des bases de donnees."
Thesis 1981-82, Institut d'Informatique, Namur

[BODA.83] F. BODART & Y. PIGNEUR
"Conception assistee des applications informatiques. 1. Etude d'opportunite et analyse conceptuelle."
MASSON, Paris 1983

[BUYS.84] M. BUYSE
"LEX, YACC, Constructeur d'arbres."
Institut d'Informatique, June 1984

[BUYS.85a] M. BUYSE
"LDF: Langage de definition de formalismes."
Institut d'Informatique, January 1985

[BUYS.85b] M. BUYSE
"Le transformateur."
Institut d'Informatique, January 1985

[CLAR.81] A. CLARINVAL
"Comprendre, Connaitre et Maitriser le Cobol Norme Ansi Cobol 1974."
Presses Universitaires de Namur 1981, 1ere edition

[DATE.81] DATE
"An Introduction to Data Base Systems."
Vol.1 Addison-Wesley, 3rd Edition, 1981

[DELC.84] B. DELCOURT & L. MOENTACK
"Contribution to the Design of a Database Workbench: Logical Record Access Evaluation."
Thesis 1983-84, Institut d'Informatique, Namur

[HAIN.81] J-L. HAINAUT
"Un modele descriptif de bases de donnes au niveau organique: le modele d'acces."
Notes de cours, Institut d'Informatique, January 1981

[HAIN.83a] J-L. HAINAUT
"ADL: un langage de description d'algorithmes."
Notes de cours, Institut d'Informatique, 1983

[HAIN.83b] J-L. HAINAUT
"Cadre de reference pour la conception de bases de donnees."
Institut d'Informatique, December 1983

[HAIN.84a] J-L. HAINAUT
"Le modele d'acces generalise."
Institut d'Informatique, January 1984

[HAIN.84b] J-L. HAINAUT
"Programmation d'application sur bases de donnees."
Institut d'Informatique, July 1984

[HAIN.85] J-L. HAINAUT
"Conception assistee des applications informatiques. 2. Conception de la base de donnees."
MASSON, Paris, 1985

[JARD.84] D.A. JARDINE
"Concepts and Terminology for the Conceptual Schema and the Information Base."
Computers & Standards 3, 1984 North-Holland

[JOHN.75] S.C. JOHNSON
"YACC, yet another compiler compiler."
Comp. Sci. Tech., Rep No 32, Bell laboratories, July 1975

[JOHN.77] S.C. JOHNSON & M.E. LESK
"Language Development Tools."
The Bell System Technical Journal, July-August 1978

[KERN.78] B.W. KERNIGHAN & D.M. RITCHIE
"The C Programming Language."
Prentice-Hall, 1978

[LESK.75] M.E. LESK & E. SCHMIDT
"LEX, a lexical analyser generator."
Comp. Sci. Tech., Rep No 39, Bell laboratories, October 1975

[MART.82] J. MARTIN
"Application development without programs."
Prentice-Hall, 1982

[MEND.80] K.S. MENDES
"Structured Systems Analysis: A Technique to Define Business Requirements."
Sloan Management Review, Summer 1980

[ZAVE.84] P. ZAVE
"The Operational Versus the Conventional Approach to Software Development."
CACM, vol.27, n.2, February 1984

[ZIMM.83] R.P. ZIMMERMAN
"Phases, Methods and Tools - A Triad of System Development."
Paper for the Third International Conference on the E-R (Entity-Relation) Approach
Standard Oil Company of California, 1983

Facultés Universitaires
Notre-Dame de la Paix
NAMUR

Institut d'Informatique

CONTRIBUTION TO A DATABASE DESIGN WORKBENCH

ADL - COBOL/GAM COMPILER

APPENDIX

Mémoire présenté par
M. Cadelli / D. Muller
en vue de l'obtention
du titre de
Licencié et Maître
en Informatique
Année académique 1984-1985

APPENDIX
****

TABLE OF CONTENTS.

SPECIFICATION OF THE FUNCTIONS OF THE G A M

****
*   *
* NOTICE *
*   *
****

We want to draw your attention to the fact that this part of the appendix gives a more recent specification of the access module functions. Changes have been done since the writing of chapter two of the thesis.

The modifications mainly concern the functions manipulating reference variables. Two functions have been added:

- the creation of a reference variable
- the deletion of a reference variable.

These functions are used to confere or retrieve the status of a reference variable to a variable of the program of type integer. To explain the notion of reference variable, we have included a general definition of this notion.

SPECIFICATION OF THE FUNCTIONS OF THE G A M

I. LIST OF THE FUNCTIONS OFFERED BY THE G A M

1. Opening of the database (OPENDB)

2. Closing of the database (CLOSEDB)

3. Access to the next instance of a record type (this includes the access to the first record of a type) (SEQ)

4. Access by key to the next instance of a record type (this includes the access to the first record matching the condition) (KEY)

5. Access to the next instance of a record type within an access path (this includes the access to the first record in the access path) (PATH)

6. Access by key to the next instance of a record type within an access path (this includes the access to the first record in the access path matching the selection condition) (KEYPATH)

7. Access by reference to an instance of a record type (DIRECT)

8. Creation of a record of a given type (CREATE)

9. Deletion of a record of a given type (DELETE)

10. Modification of some data item values of a record (the data items must not be a component of any kind of IKDs, that's to they must not be part of either an identifier, a key or an order) (MODITEM)

11. Modification of some components of IKDs (only components of an identifier, key or order may be changed) (MODIKD)

12. Transfer, attach, detach a record from one access path to another of the same type (MODPATH)

20. Comparison of two reference variables (COMPAR)

21. Annulment of a reference (NULL)

22. Assignation of a reference to another (ASSIGN)

23. Creation of a reference variable (GET)

24. Deletion of a reference variable (FREE)

Definition of the reference variables

The access to the DB is realized by means of primitive access routines using reference variables. A reference variable (in brief: a reference) designates either a database record or nothing. At this moment the reference is said to be 'null'.

e.g. : - within an access path of type CJSORD you accede to the instance of the record type ORDER following the instance denoted by the reference variable 'ORD'
- within an access path of type CJSORD you accede to the first record of type ORDER

A variable of the programming language of type integer takes the status of a reference variable by the use of the primitive function 'creation of a reference variable'.

The value of a reference variable has no meaning as an integer value. The reference variables can only be manipulated by using the primitive routines:

- Annulment of a reference
- Assignation of a reference to another
- Comparison of two reference variables

Thus one is not allowed to memorize the value of reference variables, to compare these variables or to affect a value to these variables by means of some instruction of the programming language.

N.B. The only exception concerns the parameter CURLIST[] where the value of a used reference variable must be assigned to the element CURLIST[i].

II. LIST OF THE PARAMETERS AND THEIR MEANING

SYNOPSIS:

```
BD (dbstat,opcode,getcode,recref,keycode,
    operator,pathlist,curlist,z_values)

struct  dbstat_struct *dbstat;
long int *opcode;
long int *getcode;
long int *recref;
long int *keycode;
long int *operator;
long int pathlist[];
long int curlist[];
struct sem_struct *z_values;
```

MEANING OF THE PARAMETERS

DBSTAT : code stating the result of the operation on the database
- ENCODE : contains the operation code if an error has occurred
- ERRCODE : contains the appropriate error code

OPCODE : operation code

GETCODE : code indicating if the data item values should be provided or not

RECREF : for each kind of access, it contains the reference of the previous record or a null reference if no access has been done so far; for all the database manipulations (except the creation), it contains the reference of the record involved by the operation

KEYCODE : code of the access or sort key

OPERATOR: code specifying the test operator of the selection condition

PATHLIST: list of the codes of the access path concerned by the application

CURLIST : list of the values of the references of the current origins of the access path types contained in PATHLIST

Z-VALUES:
- RECTCODE: code of the record type used in the operation
- NAME: contains the name of the database or file for the opening
- UNION OF THE RECORDS TYPES OF THE DB:
  - zone for transferring the data item values of a record from/to the interface module to/from the user; it is also used to transfer the value of a key

III. ERROR CODES OF THE ACCESS MODULE

NOTICE:

*) The errors associated to the access module and assumed specific to this module are specified below.
*) Other errors linked to the software implementing the access module may also exist provided that they have a value greater than 500. These errors are used to refine the general error diagnostic, thus facilitating a possible correction of the software.
*) For this particular database, an error code having the value of one of the errors given below plus the value 100 indicates the same type of error, but this error originates from the inferior level (internal DBMS).

err_db_ok	0	the operation has been successfully executed
err_db_nof	1	the access operation failed (record not found, DB not found)
err_db_unq	2	violation of an identifier constraint
err_db_mand	3	attempt to detach a record although the record type is a mandatory target of the access path type
err_db_nop	10	the database is not open
err_db_aop	11	the database is already open
err_invop	20	invalid operation code (operation does not exist)
err_naop	21	the required operation does not correspond to the actions allowed on the schema
err_key	22	invalid KEYCODE (the key is inexistent in the schema)
err_path	23	invalid access path type code (inexistent in the schema)
err_rect	24	invalid record type code (inexistent in the schema)
err_comp	25	the test operator is not implemented by the DBMS
err_get	26	invalid GETCODE (the specified value is undefined in the DBMS)
err_recref	27	invalid RECREF (error on the record reference)
err_curlist	29	CURLIST contains at least one record reference which is not origin of an access path of the type specified in the corresponding element of PATHLIST

err_eql 30 both reference variables reference a same record

err_dif 31 both variables reference a different record

err_db_sch 40 the name of the database schema is invalid

N.B. Two error codes are used in the functions which manipulate references to database records with a particular meaning; it is explained in the description of these operations.

1. OPENING OF THE DATABASE

IN

* OPCODE (OPENDB value = 1)

OUT

* DBSTAT (contains operation and error code)

FUNCTIONS

* opens (update exclusive) all the files necessary to the database

ERROR CODES

err_db_ok
err_db_aop
err_db_sch
err_invop

2. CLOSING OF THE DATABASE

IN

* OPCODE (CLOSEDB value = 2)

OUT

* DBSTAT (contains operation and error code)

FUNCTIONS

* closes all the files of the database

ERROR CODES

err_db_ok
err_db_nop
err_invop

3. SEQUENTIAL ACCESS

IN

* OPCODE (SEQ value = 3)
* GETCODE (value=1)
* RECTCODE (contains the code of the record type concerned by the access)
* RECREF (contains the reference of the previously acceded record or a null reference if no previous access)
* KEYCODE (contains the key of the sorting key)

OUT

* DBSTAT (contains operation and error code)
* RECREF (contains the reference of the record for which the access has been required)
* Z-VALUES (the list of the data item values of that record)

FUNCTIONS

* provides the next instance of the record type specified in RECTCODE; it is the record that immediately follows the instance referenced by RECREF (if RECREF is a null reference, it is the first record) among all the records of that type according to a sequential order or according to a sorted order whose code is given in KEYCODE;

* provides the data item values of the record in Z-VALUES

* returns the reference of the acceded record in RECREF

ERROR CODES

err_db_ok
err_db_nof
err_db_nop
err_invop
err_key
err_rect
err_get
err_recref

4. ACCESS ON A KEY.

IN

* OPCODE (KEY value = 4)
* GETCODE (value = 1)
* RECTCODE (contains the code of the record type concerned by the access)
* RECREF (contains the reference of the previously acceded record or a null reference if no previous access)
* KEYCODE (contains the code of the access key)
* OPERATOR (value = 0: exact match)
* Z-VALUES (contains the value of the key that will be used to find the record(s) with respect to the operator)

OUT

* DBSTAT (contains operation and error code)
* RECREF (contains the reference of the record for which the access has been required)
* Z-VALUES (contains the list of the data item values of that record)

FUNCTIONS

* provides an instance (its reference is returned in RECREF) of the record type specified in RECTCODE that satisfies the given condition: the value of the data items forming the access key have to match the value specified in Z-VALUES (with respect to the test operator). The record provided will be the first to follow the instance referenced by RECREF, or if RECREF is a null reference, it will be the first record meeting the test condition.

* provides the data item values of that record in Z-VALUES

ERROR CODES

err_db_ok
err_db_nof
err_db_nop
err_invop
err_key
err_comp
err_rect
err_get
err_recref

5. SEQUENTIAL ACCESS WITHIN ACCESS PATH

IN

* OPCODE (PATH value = 5)
* GETCODE (value = 1)
* RECREF (contains the reference of the previously acceded record or a null reference if no previous access)
* PATHLIST[0] (contains the code of the access path type which is used in this access)
* CURLIST[0] (contains the value of the reference of the current origin of the specified access path type)
* KEYCODE (contains the key of the sorting key)

OUT

* DBSTAT (contains operation and error code)
* RECTCODE (contains the type of the record for which the access has been required)
* RECREF (contains the reference of the record for which the access has been required)
* Z-VALUES (contains the list of the data item values of that record)

FUNCTIONS

* provides the record that follows the instance referenced by RECREF (if RECREF is a null reference, it is the first record) within the access path whose type is specified in PATHLIST and whose origin is given in CURLIST

* provides the data item values of that record in Z-VALUES

* the order in which you will get the instances in the sequential access is a particular one: sorted according to the key whose code is given in KEYCODE

ERROR CODES

err_db_ok
err_db_nof — N.B. If RECREF references a record which does not belong to the access path identified by its type (PATHLIST[0]) and by its origin record (CURLIST[0]), it is assumed that no record has been found.
err_db_nop
err_invop
err_key
err_get
err_recref
err_path
err_curlist
err_naop

6. ACCESS WITH KEY WITHIN ACCESS PATH

IN

* OPCODE (KEYPATH value = 6)
* GETCODE (value = 1)
* RECTCODE (contains the code of the record type concerned by the access)
* RECREF (contains the reference of the previously acceded record or a null reference if no previous access)
* PATHLIST[0] (contains the code of the access path type which is used in this access)
* CURLIST[0] (contains the value of the reference of the current origin of the specified access path type)
* KEYCODE (contains the code of the access key)
* OPERATOR (value = 0: exact match)
* Z-VALUES (contains the value of the key that will be used to find the record(s) with respect to the operator)

OUT

* DBSTAT (contains operation and error code)
* RECREF (contains the reference of the record for which the access has been required)
* Z-VALUES (contains the list of the data item values of that record)

FUNCTIONS

* provides an instance (its reference is returned in RECREF) of the record type specified in RECTCODE; it is the record that follows the instance referenced by RECREF (or the first record if RECREF is a null reference) within the access path identified by PATHLIST and CURLIST and that satisfies the given condition: the value of the data items forming the access key must match the value specified in Z-VALUES (according to the specified test operator)

* provides the data item values of that record in Z-VALUES

* returns DBSTAT

ERROR CODES
++++++++++++++

err_db_ok
err_db_nof N.B. If RECREF references a record which does not belong to the access path identified by its type (PATHLIST[0]) and by its origin record (CURLIST[0]), it is assumed that no record has been found.
err_db_nop
err_invop
err_key
err_comp
err_rect
err_get
err_recref
err_patn
err_curlist
err_naop

7. ACCESS BY REFERENCE TO A INSTANCE OF A RECORD TYPE

IN
++++++

* OPCODE (DIRECT value = 7)
* GETCODE (value = 1)
* RECREF (contains the reference of the record to be acceded)
* RECTCODE (contains the code of the record type concerned by the access or INDFN if it is unknown)

OUT
++++++

* DBSTAT (contains operation and error code)
* RECTCODE (contains the code of the type of the record acceded)
* Z-VALUES (contains the list of the data item values of the record acceded)

ERROR CODES
++++++++++++++

err_db_ok
err_db_noo
err_invop
err_rect
err_recref
err_get

8. CREATION OF A RECORD

IN

* OPCODE (CREATE value = 8)
* RECTCODE (contains the code of the type of the record to be created)
* Z-VALUES (contains the list of the data item values of the record to be created)
* PATHLIST (contains the list of the codes of the access path types the new record has to be attached to)
* CURLIST (contains the list of the values of the references of the current origins of the specified access path types)

OUT

* DBSTAT (contains operation and error code)
* RECREF (contains the reference of the newly created record)

FUNCTIONS

If all the pieces of information communicated through the various parameters garantee that the integrity of the database will not be affected, this function :

* creates an instance of the record type specified in RECREF, having as data item values the values contained in Z-VALUES
* returns the reference of this record in RECREF
* attaches the record to all the access paths identified through the type given in PATHLIST and the current origin provided in CURLIST
* returns DBSTAT stating the effect of the operation

ERROR CODES

err_db_ok
err_db_nop
err_db_una
err_invop
err_rect
err_recref
err_path
err_curlist

9. DELETION OF A RECORD

IN

* OPCODE (DELETE value = 9)
* RECTCODE (contains the code of the type of the record to be deleted)
* RECREF (contains the reference of the record to be deleted)

OUT

* DBSTAT (contains operation and error code)

FUNCTIONS

* detaches the record of the type specified in RECTCODE and referenced by RECREF from all the access paths in which it is a target

* from the access paths of which it is origin, detaches all the optional targets and applies the whole process (that's to say the first three functions listed here) to all the mandatory targets

* deletes the record from the database

* returns DBSTAT

N.B. All the references of the deleted records are destroyed.

ERROR CODES

err_db_ok
err_db_nop
err_invop
err_rect
err_recref

10. MODIFICATION OF ITEMS

IN

* OPCODE (MODITEM value = 10)
* RECTCODE (contains the code of the type of the record to be modified)
* RECREF (contains the reference of the instance of the record type that is going to be modified)
* Z-VALUES (contains the list of the data item values of the record that will be modified)

OUT

* DbSTAT (contains operation and error code)

FUNCTIONS

* assigns to the data items (not part of any IKO) of the record whose reference is specified in RECREF and whose type is specified in RECTCODE, the new data values contained in Z-VALUES. In fact, Z-VALUES contains all the data item values of the record involved (that's to say, that it has the structure of the record type), but only those which are no components of some kind of IKO may be modified: even if there were changed values for those which are part of an IKO, these values would not be taken into consideration.

* returns DbSTAT

ERROR CODE

err_db_ok
err_db_nop
err_invop
err_rect
err_recref

11. MODIFICATION OF IKOs

IN

* OPCODE (MODIKO value = 11)
* RECTCODE (contains the code of the type of the record to be modified)
* RECREF (contains the reference of the instance of the record type that is going to be modified)
* Z-VALUES (contains the list of the data item values of the record that will be modified)

OUT

* DBSTAT (contains operation and error code)

FUNCTIONS

* assigns to the data items, part of an IKO and associated to the record referenced by RECREF the new data values contained in Z-VALUES (RECTCODE specifies the type). In fact, Z-VALUES contains all the data item values of the record involved (that's to say that it has the structure of the record type), but only those which are components of some kind of IKOs may be modified; even if there were changed values for the other items, these values would not be taken into consideration.

* returns DBSTAT

ERROR CODES

err_db_ok
err_db_nop
er_db_unq
err_invop
err_rect
err_recref

12. ATTACH,DETACH,TRANSFERT

IN

* OPCODE (MODPATH value = 12)
* RECREF (contains the reference of the record to be attached, detached or transferred)
* PATHLIST[0] (contains the code of the access path type implied in the operation)
* CURLIST[0] (- if the functional class of the access path type is many-to-many: contains the reference of the origin record to which the record referenced by RECREF is linked to (as a target) by an access path of the specified type; contains a null reference if the record does not yet belong to any access path of the specified type
  - if the functional class of the access path type is one-to-many: is not used because the origin record to which the record referenced by RECREF is linked to (as a target) by an access path of the specified type is known implicitly)

* CURLIST[1] (contains the reference of the record to which the record referenced by RECREF will be attached to as a target or the reference to any record)

OUT

* DBSTAT (contains the operation and error code)

FUNCTIONS

* detaches the record referenced by RECREF from the access path of which the origin record is either specified explicitly in CURLIST[0] (many-to-many) or given implicitly (one-to-many)

* attaches the same record to the access path having as origin the record specified by CURLIST[1], provided that it references a record

* if CURLIST[1] is a null reference, the record referenced by RECREF does no longer belong to any access path of the specified type

ERROR CODES

err_db_ok
err_db_nop
err_db_unq
err_invop
err_rect
err_recref
err_curlist

20. COMPARISON OF TWO REFERENCE VARIABLES

IN

* OPCODE (COMPARE value = 20)
* RECREF (contains one of the references to be compared)
* CURLIST[0] (contains the other reference of the comparison)

OUT

* DBSTAT (contains the operation and error code)

FUNCTIONS

* if both references reference the same record , return err_eql
* if they reference two different records in the database, return err_dif

ERROR CODES

err_eql
err_dif
err_curlist (if at least one of both variables is not a reference variable)
err_invop

21. ANNULMENT OF A REFERENCE

IN

* OPCODE (NULL value 21)
* RECREF (contains the reference to be reinitialized)

OUT

* DBSTAT (contains operation and error code)

FUNCTIONS

* reinitializes the reference in such a way that it no longer references a database record

ERROR CODES

err_db_ok
err_recref (if the passed variable is not a reference variable)
err_invop

22. ASSIGNATION OF A REFERENCE TO ANOTHER

IN

* OPCODE (ASSIGN value = 22)
* RECREF (contains the target reference variable)
* CURLIST[0] (contains the origin reference variable)

OUT

* DBSTAT (contains operation and error code)

FUNCTIONS

* after the assignation, the target reference variable references the same record as the origin reference variable: if the origin reference variable addresses no article, so does the target reference

ERROR CODES

err_db_ok
err_curlist (if at least one of the two variables is not a reference variable)
err_invop

23. CREATION OF A REFERENCE VARIABLE

IN

* OPCODE (GET value = 23)
* RECREF (contains a variable)

OUT

* OBSTAT (contains operation and error code)

FUNCTIONS

* after the call the variable RECREF takes the status of a reference variable

ERROR CODES

err_db_ok
err_recref
err_invop

24. DELETION OF A REFERENCE VARIABLE

IN

* OPCODE (FREE value = 24)
* RECREF (contains a variable)

OUT

* DBSTAT (contains operation and error code)

FUNCTIONS

* after the call the variable RECREF has lost its status of a reference variable

ERROR CODES

err_db_ok
err_recref
err_invop

SYNTACTICAL AND LEXICAL DEFINITION OF THE SUBSET-ADL

```
**********************************************************
*                                                        *
*      SYNTACTICAL DEFINITION OF THE SUBSET-ADL          *
*                                                        *
**********************************************************


<algorithm_structure> ::= <algorithm_heading> <algorithm_body>.

    <algorithm_heading> ::= algorithm <name>

    <algorithm_body> ::= <intern_declaration_part>
                         <statement_part>


 <intern_declaration_part> ::= <type_declaration_part>
                               <variable_declaration_part>


 <type_declaration_part> ::= type <type_declarations> | <empty>

     <type_declarations> ::= <type_declaration>
                           | <type_declaration> ;
                               <type_declarations>

     <type_declaration> ::= <name> = <variable_type>


 <variable_declaration_part> ::= var <variable_declarations>
                                 | <empty>

     <variable_declarations> ::= <variable_declaration>
                                 | <variable_declaration> ;
                                     <variable_declarations>

     <variable_declaration> ::= <names> : <variable_type>

     <names> ::= <name>,<names> | <name>

     <variable_type> ::= <simple> | <structured> | <name>

     <simple> ::= boolean
                | string(<unsigned_integer>)
                | real
                | integer
                | numeric(<unsigned_integer>,<unsigned_integer>)

     <structured> ::= <group>
                    | <array>
                    | <items_of>
                    | <ref_of>

     <group> ::= group <field_list> end
```

```
<field_list> ::= <variable_declaration> <field_list>
                 | <variable_declaration>

<array> ::= array [ <index_lst> ] of <component_type>

<index_lst> ::= <index>,<index>,<index>
                | <index>,<index>
                | <index>

<index> ::= <unsigned_integer>

<component_type> ::= <simple>
                     | <ref_of>
                     | <items_of>
                     | <name>

<items_of> ::= items_of <name>

<ref_of> ::= ref of <name> | ref of RECORD
```

```
<statement_part> ::= begin <statements> end

<statements> ::= <statement> ; <statements>
                 | <statement>

<statement> ::= <ass_st>
                | <for_st>
                | <if_st>
                | <while_st>
                | <db_mod>
                | <next_st>
                | <exit_st>
                | <call_st>
                | <return_st>
                | <dummy>

    <dummy> ::= <empty>

    <ass_st> ::= <variable> := <assign_expression>

    <for_st> ::= for <variable> := <collection_expression> do
                     <statements> endfor

        <collection_expression> ::= <range> | <DB_object_set>

        <range> ::= <range_expr>..<range_expr>

        <range_expr> ::= <unsigned_integer> | <name>

    <if_st> ::= if <general_expression> then <statements> endif
              | if <general_expression> then <statements>
                   else <statements> endif

    <while_st> ::= while <general_expression> do <statements>
                       endwhile
```

```
<next_st> ::= next <name> | next

<exit_st> ::= exit <name> | exit

<call_st> ::= <proc_name>
            | <proc_name> ( <param_list> )

<return_st> ::= return
              | return ( <param_list> )

<param_list> ::= <variable>,<param_list> | <variable>

<db_mod> ::= <modif>
           | <creat>
           | <del>

<creat> ::= create <name> := <name> <creat_conds>

    <creat_conds> ::= <creat_cond>
                    | (<conditions>)

    <conditions> ::= <creat_cond> and <conditions>
                   | <creat_cond>

    <creat_cond> ::= <attach_cond>
                   | <item_cond>

<modif> ::= modify <name> <modif_conds>

    <modif_conds> ::= <attach_cond>
                    | <items_conds>
                    | <detach_cond>

    <items_conds> ::= <item_cond>
                    | (<conds>)

    <conds> ::= <item_cond>
              | <item_cond> and <conds>

    <item_cond> ::= <relation_operator> <name> =
                        <arithmetic_expression> )

    <attach_cond> ::= <relation_operator> <name> )

    <detach_cond> ::= <relation_operator> <unsigned_integer>
                          <name> )

<del> ::= delete <name>
```

```
<assign_expression> ::= <arithmetic_expression>
                      | <general_expression>

<general_expression> ::= <general_factor> or <general_expression>
                       | <general_factor>

<general_factor> ::= <general_term> and <general_factor>
                   | <general_term>

<general_term> ::= not <general_primary>
                 | <general_primary>

<general_primary> ::= <test_expression>
                    | (<general_expression>)

<test_expression> ::= <arithmetic_expression> <test_operator>
                      <arithmetic_expression>


<arithmetic_expression> ::= <factor> <adding>
                                  <arithmetic_expression>
                          | <adding> <arithmetic_expression>
                          | <factor>

    <factor> ::= <term> <multiplying> <factor>
               | <term>

    <term> ::= <primary> ^ <primary>
             | <primary>

    <primary> ::= <string>
                | <unsigned_number>
                | <logical_value>
                | <variable>
                | <call_st>
                | <nullref>
                | <DS_object_set>
                | ( <arithmetic_expression> )

    <unsigned_number> ::= <unsigned_integer>
                        | <unsigned_real>

    <adding> ::= + | -

    <multiplying> ::= * | /

    <nullref> ::= ()
```

```
<DB_object_set> ::= <name> <predicate>

    <predicate> ::= ( <coll_cond_term> ) and ( <predicate> )
                  | <coll_cond_term>

    <coll_cond_term> ::= not <coll_cond_primary>
                       | <coll_cond_primary>

    <coll_cond_primary> ::= <relation_condition>
                          | ()
                          | ( <predicate> )

    <relation_condition> ::= <relation_operator> <name> )
                           | <relation_operator> <belonging_cond>

    <relation_operator> ::= ( <name> :
                          | ( :

    <belonging_cond> ::= <name> <test_operator>
                           <arithmetic_expression>

-------------------------------------------------------------------
-------------------------------------------------------------------

<variable> ::= <class_var>
             | <var_items>

<class_var> ::= <hierar_variable>
              | <non_hierar_variable>

    <non_hierar_variable> ::= <name>
                            | <subscripted_variable>

    <subscripted_variable> ::= <name> [ <sub_index> ]

    <sub_index> ::= <arithmetic_expression>,
                        <arithmetic_expression>,
                            <arithmetic_expression>
                  | <arithmetic_expression>,
                        <arithmetic_expression>
                  | <arithmetic_expression>

    <hierar_variable> ::= <non_hierar_variable>.<class_var>

<var_items> ::= (<name>).<class_var>
              | (<name>).FILE
              | (<name>).TYPE

-------------------------------------------------------------------
-------------------------------------------------------------------
```

```
*****************************************
*   LEXICAL DEFINITION OF SUBSET-ADL    *
*****************************************

<name> ::= <name> <letter>
         | <name> <digit>
         | <name> _
         | <letter>

<proc_name> ::= <name>!

<unsigned_integer> ::= <unsigned_integer> <digit>
                     | <digit>

    <decimal_number> ::= <unsigned_integer>.<unsigned_integer>

    <sign> ::= + | -

    <exponent_part> ::= e <sign> <unsigned_integer>
                      | e <unsigned_integer>

    <unsigned_real> ::= <decimal_number> <exponent_part>
                      | <decimal_number>

<string> ::= ' <any sequence of basic symbols not containing '> '

-------------------------------------------------------------------

<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<empty> ::= ' '

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<logical_value> ::= true | false

<test_operator> ::= < | > | <= | >= | <> | =
```

SYNTACTICAL AND LEXICAL DEFINITION OF ADL

```
*****************************************
*                                       *
*     SYNTACTICAL DEFINITION OF ADL     *
*                                       *
*****************************************

<algorithm_structure> ::= <algorithm_heading> <algorithm_body>.

    <algorithm_heading> ::= algorithm <name>

    <algorithm_body> ::= <intern_declaration_part>
                         <statement_part>


<intern_declaration_part> ::= <type_declaration_part>
                              <variable_declaration_part>


<type_declaration_part> ::=  type <type_declarations> | <empty>

    <type_declarations> ::= <type_declaration>
                          | <type_declaration> ;
                              <type_declarations>

    <type_declaration> ::= <name> = <variable_type>


<variable_declaration_part> ::= var <variable_declarations>
                              | <empty>

    <variable_declarations> ::= <variable_declaration>
                              | <variable_declaration> ;
                                  <variable_declarations>

    <variable_declaration> ::= <names> : <variable_type>

    <names> ::= <name>,<names> | <name>

    <variable_type> ::= <simple> | <structured> | <name>

    <simple> ::= boolean
               | string(<unsigned_integer>)
               | real
               | integer
               | numeric(<unsigned_integer>,<unsigned_integer>)

    <structured> ::= <group>
                   | <array>
                   | <items_of>
                   | <ref_of>
                   | <list_of>

    <group> ::= group <field_list> end

    <field_list> ::= <variable_declaration> <field_list>
                   | <variable_declaration>
```

```
<array> ::= array [ <index_lst> ] of <component_type>

<index_lst> ::= <index>,<index_lst>
              | <index>

<index> ::= <unsigned_integer>

<component_type> ::= <simple>
                   | <items_of>
                   | <ref_of>
                   | <name>

<items_of> ::= items_of <name>

<list_of> ::= list of <component_type>
            | list [<unsigned_integer>] of <component_type>

<ref_of> ::= ref of <name> | ref of RECORD
```

```
<statement_part> ::= begin <statements> end

<statements> ::= <statement> ; <statements>
               | <statement>

<statement> ::= <ass_st>
              | <for_st>
              | <if_st>
              | <while_st>
              | <db_mod>
              | <next_st>
              | <exit_st>
              | <call_st>
              | <return_st>
              | <dummy>

<dummy> ::= <empty>

<ass_st> ::= <variable> := <assign_expression>

<for_st> ::= for <variable> := <for_list> do <statements>
                 <if_no_st> endfor

    <for_list> ::= <collection_expression> <order>
                 | <collection_expression>

    <order> ::= order <order_keys>

    <order_keys> ::= <name> <ad> , <order_keys>
                   | <name> <ad>

    <ad> ::= ascending | descending | <empty>

    <if_no_st> ::= if_no <variable> then <statements> | <empty>
```

```
<if_st> ::= if <general_expression> then <statements> endif
          | if <general_expression> then <statements>
               else <statements> endif

<while_st> ::= while <general_expression> do <statements>
                    endwhile

<next_st> ::= next <name> | next

<exit_st> ::= exit <name> | exit

<call_st> ::= <name>
            | <name> ( <param_list> )

<return_st> ::= return
              | return ( <param_list> )

<param_list> ::= <primary>,<param_list> | <primary>

<db_mod> ::= <modif>
           | <creat>
           | <del>

<modif> ::= modify <name> <modif_conds>

    <modif_conds> ::= <modif_cond> | (<conds>)

    <conds> ::= <modif_cond>
              | <modif_cond> and <conds>

    <modif_cond> ::= <detach_cond>
                   | <attach_cond>
                   | <item_cond>

    <detach_cond> ::= <relation_operator> <unsigned_integer>
                          <name>)

    <attach_cond> ::= <relation_operator> <name>)
                    | <relation_operator> <DB_object_set>)

    <item_cond> ::= <relation_operator> <name> =
                        <arithmetic_expression>)

<creat> ::= create <name> := <name> <creat_conds>

    <creat_conds> ::= <creat_cond> | (<conditions>)

    <conditions> ::= <creat_cond>
                   | <creat_cond> and <conditions>

    <creat_cond> ::= <attach_cond> | <item_cond>


<del> ::= delete <name>
        | delete <name> <relation_operator> <unsigned_integer>
              <name> )
```

```
<assign_expression> ::= <arithmetic_expression>
                      | <general_expression>


<general_expression> ::= <general_factor> or <general_expression>
                       | <general_factor>

<general_factor> ::= <general_term> and <general_factor>
                   | <general_term>

<general_term> ::= not <general_primary>
                 | <general_primary>

<general_primary> ::= <test_expression>
                    | (<general_expression>)

<test_expression> ::= <arithmetic_expression> <test_operator>
                        <arithmetic_expression>


<arithmetic_expression> ::= <factor> <adding>
                                <arithmetic_expression>
                          | <adding> <arithmetic_expression>
                          | <factor>

    <factor> ::= <term> <multiplying> <factor>
               | <term>

    <term> ::= <primary> ^ <primary>
             | <primary>

    <primary> ::= <string>
                | <unsigned_number>
                | <logical_value>
                | <variable>
                | <call_st>
                | <nullref>
                | <collection_expression>
                | ( <arithmetic_expression> )

    <unsigned_number> ::= <unsigned_integer>
                        | <unsigned_real>

    <adding> ::= + | -

    <multiplying> ::= * | /

    <nullref> ::= ()
```

```
<collection_expression> ::= <range>
                          | <list>
                          | <DB_object_set>

<range> ::= <arithmetic_expression>..<arithmetic_expression>
          | *..<arithmetic_expression>
          | <arithmetic_expression>..*
          | *..*

<list> ::= { <list_enumeration> }

    <list_enumeration> ::= <list_element> , <list_enumeration>
                         | <list_element>

    <list_element> ::= <assign_expression>

<DB_object_set> ::= <variable> <predicate>

    <predicate> ::= ( <coll_cond_factor> ) or ( <predicate> )
                  | <coll_cond_factor>

    <coll_cond_factor> ::= ( <coll_cond_term> ) and
                             ( <coll_cond_factor> )
                         | <coll_cond_term>

    <coll_cond_term> ::= not <coll_cond_primary>
                       | <coll_cond_primary>

    <coll_cond_primary> ::= <relation_condition>
                          | ()
                          | ( <predicate> )

    <relation_condition> ::= <relation_operator> <co> <variable>)
                           | <relation_operator> <co>
                               <DB_object_set>)
                           | <relation_operator> <co>
                               <belonging_cond>)

    <relation_operator> ::= ( <name> : <co>
                          | ( : <co>

    <co> ::= [<cardinal>] | <ordinal> | <empty>

    <cardinal> ::= <range> | * | <unsigned_integer> | <name>

    <ordinal> ::= £ <cardinal>

    <belonging_cond> ::= <variable> <test_operator>
                           <arithmetic_expression>
```

```
<variable> ::= <class_var>
             | <var_items>
             | <name>//<name>

<class_var> ::= <hierar_variable>
              | <non_hierar_variable>

    <non_hierar_variable> ::= <name>
                            | <subscripted_variable>

    <subscripted_variable> ::= <name> [ <subscript_list> ]

    <subscript_list> ::= <arithmetic_expression>,<subscript_list>
                       | <arithmetic_expression>

    <hierar_variable> ::= <non_hierar_variable>.<class_var>

    <var_items> ::= (<name>).<class_var>
                  | (<name>).FILE
                  | (<name>).TYPE
```

```
    **********************************
    *   LEXICAL DEFINITION OF ADL    *
    **********************************

<name> ::= <name> <letter>
         | <name> <digit>
         | <name> _
         | <letter>

<unsigned_integer> ::= <unsigned_integer> <digit> | <digit>

    <decimal_number> ::= <unsigned_integer>.<unsigned_integer>

    <sign> ::= + | -

    <exponent_part> ::= e <sign> <unsigned_integer>
                      | e <unsigned_integer>

    <unsigned_real> ::= <decimal_number> <exponent_part>
                      | <decimal_number>

<string> ::= ' <any sequence of basic symbols not containing '> '
```

```
<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<empty> ::= ' '

<logical_value> ::= true | false

<test_operator> ::= < | > | <= | >= | <> | = | in | not_in
```

DESCRIPTION OF THE CONCEPTUAL SCHEMA OF THE WORKBENCH DATABASE IN THE META FORMALISM

```
£££££££££££££££££££££££££££££££££££££££££££££££
£                                                 £
£                SYSTEM-SECTION                   £
£                                                 £
£££££££££££££££££££££££££££££££££££££££££££££££

SYSTEM LDD/ADD;
   LANGUAGE DDL;
£                  >>>>> DATA DEFINITION LANGUAGE

   ANALYZER DDA;
£                  >>>>> DATA DEFINITION ANALYSER

£££££££££££££££££££££££££££££££££££££££££££££££
£
£ Predefined part for META M1.2
£
£ Place this at the beginning of the META definition.
£
£ The predefined parts are used for giving forms for
£ the predefined statements (Section header, synonym,
£ property and text statements).
£ The FORM statements in the predefined parts may be
£ changed to give different syntax for these statements.
£
£ Predefined form part names are
£
£    DEFINED-OBJECT       object to be defined
£    DEFINED-TYPE         object type
£    SYNONYM-TYPE         synonym name
£    TEXT-TYPE            text type name
£    PROPERTY-TYPE        property
£    PROPERTY-VALUE-TYPE  value for the property
£
£ Note that the "initialized" meta data base PREDEF.MT
£ already contains this information, and that putting
£ predefined part into a meta data base that already
£ contains it will cause problems.
£££££££££££££££££££££££££££££££££££££££££££££££

KEYWORD DEFINE;
   SYNONYMS DEF;

KEYWORD SYNONYM;
   SYNONYMS SYN,SYNONYMS;

PREDEFINED-STATEMENT PREDEFINED-SECTION-STATEMENT;
   FORM DEFINE DEFINED-TYPE ( DEFINED-OBJECT : ,);

PREDEFINED-STATEMENT PREDEFINED-SYNONYM-STATEMENT;
   FORM SYNONYM ( SYNONYM-TYPE : ,) ;

PREDEFINED-STATEMENT PREDEFINED-TEXT-STATEMENT;
   FORM TEXT-TYPE ;

PREDEFINED-STATEMENT PREDEFINED-PROPERTY-STATEMENT;
   FORM PROPERTY-TYPE PROPERTY-VALUE-TYPE;
```

```
£
£ End of predefined part
£££££££££££££££££££££££££££££££££££££££££££££££££££

£££££££££££££££££££££££££££££££££££££££££££££££££££
£ OPTIONAL-WORD OPTIONAL-WORD OPTIONAL-WORD OPTIONAL-WORD  £
£££££££££££££££££££££££££££££££££££££££££££££££££££

OPTIONAL-WORD IN;
OPTIONAL-WORD IS;
OPTIONAL-WORD BY;
OPTIONAL-WORD TO;
OPTIONAL-WORD OF;
OPTIONAL-WORD FOR;

£££££££££££££££££££££££££££££££££££££££££££££££££££
£ NAME-CONSTANT NAME-CONSTANT NAME-CONSTANT NAME-CONSTANT  £
£££££££££££££££££££££££££££££££££££££££££££££££££££

NAME-CONSTANT NM;
   DESCRIPTION;
               none or more;
NAME-CONSTANT OM;
   DESCRIPTION;
               one or more;
NAME-CONSTANT OO;
   DESCRIPTION;
               only one;
NAME-CONSTANT NO;
   DESCRIPTION;
               none or one;
NAME-CONSTANT ORIGIN;
NAME-CONSTANT TARGET;
NAME-CONSTANT NAS;
   DESCRIPTION;
               necessary access schema;
NAME-CONSTANT PAS;
   DESCRIPTION;
               possible access schema;
NAME-CONSTANT EAS;
   DESCRIPTION;
               effective access schema;
NAME-CONSTANT FIXED;
NAME-CONSTANT VARIABLE;
NAME-CONSTANT IKO;
   DESCRIPTION;
               identifier or key or order;
NAME-CONSTANT IK;
NAME-CONSTANT I;
NAME-CONSTANT KO;
NAME-CONSTANT O;
NAME-CONSTANT IO;
NAME-CONSTANT K;
NAME-CONSTANT NUM;
   DESCRIPTION;
               numeric;
```

```
NAME-CONSTANT CHAR;
   DESCRIPTION;
               character;
NAME-CONSTANT DEC;
   DESCRIPTION;
               decomposable;
NAME-CONSTANT DB;
   DESCRIPTION;
               data base;
NAME-CONSTANT EF;
   DESCRIPTION;
               every file;
NAME-CONSTANT AP;
   DESCRIPTION;
               access path;
NAME-CONSTANT EM;
   DESCRIPTION;
               effective module;
NAME-CONSTANT PM;
   DESCRIPTION;
               predicative module;
NAME-CONSTANT ASCENDING;
NAME-CONSTANT DESCENDING;
NAME-CONSTANT YES;
NAME-CONSTANT NO;
NAME-CONSTANT RANDOM;
NAME-CONSTANT FIFO;
NAME-CONSTANT LIFO;
NAME-CONSTANT PRIOR;
NAME-CONSTANT NEXT;
NAME-CONSTANT SORTED;

£££££££££££££££££££££££££££££££££££££££££££££££££
£ OBJECTS OBJECTS OBJECTS OBJECTS OBJECTS OBJECTS OBJECTS  £
£££££££££££££££££££££££££££££££££££££££££££££££££

OBJECT ACCESS-SCHEMA;
            SYNONYMS SCH,SCHEMA;
            CODE SCHSCH 10;

   PROPERTY SCH-TYPE ;
            APPLIES ACCESS-SCHEMA;
            DOCUMENTATION;
            necessary, possible or effective access schema;
            VALUES NAS,PAS,EAS;
            CODE SCHTYP 101;

   PROPERTY SCH-NAME;
            APPLIES ACCESS-SCHEMA;
            VALUES STRING;
            CODE SCHNAM 102;

   PROPERTY SCH-CODE;
            APPLIES ACCESS-SCHEMA;
            DOCUMENTATION;
            code which identifies the schema within the DB
            it is used in the access functions;
            VALUES INTEGER 0 THRU 1000;
            CODE SCHCOD 103;
```

```
OBJECT FILE;
            SYNONYMS F;
            CODE FILFIL 11;

   PROPERTY F-NAME;
            APPLIES FILE;
            VALUES STRING;
            CODE FILNAM 104;

   PROPERTY F-CODE;
            APPLIES FILE;
            DOCUMENTATION;
            code which identifies the file within the DB
            it is used in the access functions;
            VALUES INTEGER 0 THRU 1000;
            CODE FILCOD 105;

   PROPERTY F-TYPE;
            APPLIES FILE;
            VALUES STRING;
            CODE FILTYP 106;


OBJECT RECORD-TYPE;
            SYNONYMS RT;
            CODE RECREC 12;

   PROPERTY RT-NAME;
            APPLIES RECORD-TYPE;
            VALUES STRING;
            CODE RECNAM 107;

   PROPERTY RT-CODE;
            APPLIES RECORD-TYPE;
            DOCUMENTATION;
            code which identifies the record type within the schem
            it is used in the access functions;
            VALUES INTEGER 0 THRU 1000;
            CODE RECCOD 108;

   PROPERTY RT-TYPE;
            APPLIES RECORD-TYPE;
            VALUES STRING;
            CODE RECTYP 109;


OBJECT DATA-ITEM;
            SYNONYMS ITEM;
            CODE ITEITE 13;

   PROPERTY IT-NAME;
            APPLIES DATA-ITEM;
            VALUES STRING;
            CODE ITENAM 110;
```

```
PROPERTY IT-CODE;
         APPLIES DATA-ITEM;
         DOCUMENTATION;
         code which identifies the data item within the
         record type, it is used in the access functions;
         VALUES INTEGER 0 THRU 1000;
         CODE ITECOD 111;

PROPERTY IT-SEQ;
         APPLIES DATA-ITEM;
         DOCUMENTATION;
         ordinal number of the data item
         on this level of decomposition;
         VALUES INTEGER 0 THRU 1000;
         CODE ITESEQ 112;

PROPERTY OPTIONAL;
         APPLIES DATA-ITEM;
         DOCUMENTATION;
         a data item is mandatory or optional;
         VALUES YES,NO;
         CODE ITEOPT 113;

PROPERTY REPETITIVITY;
         APPLIES DATA-ITEM;
         DOCUMENTATION;
          =0   if not repetitive
          =n   if the repetitivity is limited
               it is fixed if the data item has
                 no counter, variable otherwise
          =999 if the repetitivity is unlimited;
         VALUES INTEGER 0 THRU 999;
         CODE ITEREP 114;

PROPERTY TYPE;
         APPLIES DATA-ITEM;
         DOCUMENTATION;
         possible values : numeric, character, decomposable;
         VALUES NUM,CHAR,DEC;
         CODE ITETYP 115;

PROPERTY LENGTH;
         APPLIES DATA-ITEM;
         DOCUMENTATION;
         maximal size of the length of the possible values
         of this data item;
         VALUES INTEGER 1 THRU 100;
         CODE ITELEN 116;

PROPERTY DECIM;
         APPLIES DATA-ITEM;
         DOCUMENTATION;
         number of decimals for a data item of type numeric;
         VALUES INTEGER 0 THRU 100;
         CODE ITEDEC 117;
```

```
OBJECT PATH-TYPE;
            SYNONYMS APT;
            CODE PATPAT 14;

   PROPERTY AP-NAME;
            APPLIES PATH-TYPE;
            VALUES STRING;
            CODE PATNAM 118;

   PROPERTY AP-CODE;
            APPLIES PATH-TYPE;
            DOCUMENTATION;
            code which identifies the path type within the schema
            it is used in the access functions;
            VALUES INTEGER 0 THRU 1000;
            CODE PATCOD 119;


OBJECT IDENTIFIER-KEY-ORDER;
            SYNONYMS IKO;
            CODE IKOIKO 15;

   PROPERTY IKO-CODE;
            APPLIES IDENTIFIER-KEY-ORDER;
            DOCUMENTATION;
            code which identifies the iko within the record type
            it is used in the access modules;
            VALUES INTEGER 0 THRU 1000;
            CODE IKOCOD 120;

   PROPERTY I-K-O;
            APPLIES IDENTIFIER-KEY-ORDER;
            DOCUMENTATION;
            indicates if the IKO represents
            an identifier, a key or an order;
            VALUES IKO,IK,IO,I,KO,K,O;
            CODE IKOIOE 121;

   PROPERTY NUM-GLOBAL;
            APPLIES IDENTIFIER-KEY-ORDER;
            DOCUMENTATION;
            0 if the identifier, access key or order
            only apply to one record type,
            otherwise all the IKOs with a same number of
            global define an identifier, access key or order
            for several record type;
            VALUES INTEGER 0 THRU 1000;
            CODE IKONUM 122;

   PROPERTY GLOBAL-ORDER;
            APPLIES IDENTIFIER-KEY-ORDER;
            DOCUMENTATION;
            defines the global order;
            VALUES RANDOM,FIFO,LIFO,PRIOR,NEXT,SORTED;
            CODE IKOGLO 123;
```

```
    PROPERTY SEQ-GLOBAL;
             APPLIES IDENTIFIER-KEY-ORDER;
             DOCUMENTATION;
             defines the ordering within one
             record type in a global order;
             VALUES INTEGER 0 THRU 1000;
             CODE IKOSEQ 124;

    PROPERTY SIMPLE-ORDER;
             APPLIES IDENTIFIER-KEY-ORDER;
             DOCUMENTATION;
             defines the order in a record type;
             VALUES RANDOM,FIFO,LIFO,PRIOR,NEXT,SORTED;
             CODE IKOSIM 125;

    PROPERTY DUPLICATES;
             APPLIES IDENTIFIER-KEY-ORDER;
             DOCUMENTATION;
             indicates if duplicates are allowed
             and how they are ordered;
             VALUES NO,RANDOM,FIFO,LIFO,PRIOR,NEXT;
             CODE IKODUP 126;

    PROPERTY REF-SET;
             APPLIES IDENTIFIER-KEY-ORDER;
             DOCUMENTATION;
             reference set of the IKO is
             db : all the database
             ef : each file containing the record taken separately
             ap : access path associated to the iko;
             VALUES DB,EF,AP;
             CODE IKOSET 127;


OBJECT MODULE;
             SYNONYMS MOD;
             CODE MODMOD 16;

    PROPERTY MO-NAME;
             APPLIES MODULE;
             VALUES STRING;
             CODE MODNAM 128;

    PROPERTY MO-CODE;
             APPLIES MODULE;
             DOCUMENTATION;
             code which identifies the module within the DB
             VALUES INTEGER 0 THRU 1000;
             CODE MODCOD 129;

    PROPERTY MO-TYPE;
             APPLIES MODULE;
             DOCUMENTATION;
             effective,predicative;
             VALUES EM,PM;
             CODE MODTYP 130;
```

```
£££££££££££££££££££££££££££££££££££££££££££££££££££
£ RELATIONS    RELATIONS    RELATIONS    RELATIONS          £
£££££££££££££££££££££££££££££££££££££££££££££££££££

RELATION file-of-rel;
            PARTS              part-1,part-2;
            COMBINATION        part-1 SCHEMA
                  WITH         part-2 FILE;
            CONNECTIVITY       ONE  part-1
                               MANY part-2;
            CONNECTION-TYPE    S2;
            STORED             part-1 1,
                               part-2 2;
            CODE      RELFIL   301;

STATEMENT file-part-1-stm;
            USED part-1 file-of-rel;
            FORM COLLECTION OF (part-2:,);

STATEMENT file-part-2-stm;
            USED part-2 file-of-rel;
            FORM COLLECTED IN SCHEMA (part-1:,);

£-----------------------------------------------------------

RELATION from-rel;
            PARTS              part-1,part-2;
            COMBINATION        part-1 ACCESS-SCHEMA
                  WITH         part-2 RECORD-TYPE;
            CONNECTIVITY       ONE  part-1
                               MANY part-2;
            CONNECTION-TYPE    S2;
            STORED             part-1 1,
                               part-2 2;
            CODE      RELFRO   302;

STATEMENT from-part-1-stm;
            USED part-1 from-rel;
            FORM COLLECTION OF (part-2:,);

STATEMENT from-part-2-stm;
            USED part-2 from-rel;
            FORM COLLECTED IN SCHEMA (part-1:,);

£-----------------------------------------------------------

RELATION in-rel;
            PARTS              part-1,part-2;
            COMBINATION        part-1 FILE
                  WITH         part-2 RECORD-TYPE;
            CONNECTIVITY       MANY part-1,
                                    part-2;
            CONNECTION-TYPE    S1;
            STORED             part-1 1,
                               part-2 2;
            CODE      RELAIN   303;
```

```
STATEMENT in-part-1-stm;
            USED part-1 in-rel;
            FORM COLLECTION OF (part-2:,);

STATEMENT in-part-2-stm;
            USED part-2 in-rel;
            FORM COLLECTED IN FILE (part-1:,);

£-------------------------------------------------------------

RELATION contains-rel;
            PARTS              part-1,part-2;
            COMBINATION        part-1 RECORD-TYPE,
                                      DATA-ITEM
                    WITH       part-2 DATA-ITEM;
            CONNECTIVITY       ONE    part-1
                               MANY   part-2;
            CONNECTION-TYPE    S2;
            STORED             part-1 1,
                               part-2 2;
            CODE       RELCON  304;

STATEMENT contains-rel-part-1-stm;
            USED part-1 contains-rel;
            FORM CONTAINS (part-2:,);

STATEMENT contains-rel-part-2-stm;
            USED part-2 contains-rel;
            FORM IS CONTAINED IN (part-1:,);

£-------------------------------------------------------------

RELATION Counts-rel;
            PARTS              part-1,part-2;
            COMBINATION        part-1 DATA-ITEM
                    WITH       part-2 DATA-ITEM;
            CONNECTIVITY       ONE  part-1
                               MANY part-2;
            CONNECTION-TYPE    S2;
            STORED             part-1 1,
                               part-2 2;
            CODE       RELCOU  305;

STATEMENT DATA-ITEM-counter-part-1-stm;
            USED part-1 Counts-rel;
            FORM COUNTER OF (part-2:,);

STATEMENT DATA-ITEM-counter-part-2-stm;
            USED part-2 Counts-rel;
            FORM COUNTED BY part-1;

£-------------------------------------------------------------
```

```
RELATION concerns-rel;
            PARTS               part-1,part-2;
            COMBINATION         part-1 RECORD-TYPE
                    WITH        part-2 IDENTIFIER-KEY-ORDER;
            CONNECTIVITY        ONE  part-1
                                MANY part-2;
            CONNECTION-TYPE     S2;
            STORED              part-1 1,
                                part-2 2;
            CODE        RELCER  306;

STATEMENT iko-concerns-part-2-stm;
            USED part-2 concerns-rel;
            FORM CONCERNS(part-1:,);

STATEMENT iko-concerns-part-1-stm;
            USED part-1 concerns-rel;
            FORM HAS IKO (part-2:,);

£-----------------------------------------------------------

RELATION component-rel;
            PARTS               part-1,part-2,part-3,part-4;
            COMBINATION         part-1 DATA-ITEM,PATH-TYPE
                    WITH        part-2 IDENTIFIER-KEY-ORDER
                    WITH        part-3 VALUE-FOR DIRECTION
                    WITH        part-4 VALUE-FOR SEQNUM;
            CONNECTIVITY        MANY part-1,
                                     part-2,
                                     part-3,
                                     part-4;
            CONNECTION-TYPE     F1;
            STORED              part-1 1,
                                part-2 2,
                                part-3 3,
                                part-4 4;
            CODE        RELCOM  307;

STATEMENT component-part-1-stm;
            USED part-1 component-rel;
            FORM IS part-4 TH COMPONENT OF part-2 WITH part-3
                ORDER;

STATEMENT component-part-2-stm;
            USED part-2 component-rel;
            FORM HAS part-1 AS part-4 TH COMPONENT WITH part-3
                ORDER;

£-----------------------------------------------------------
```

```
RELATION member-rel;
            PARTS              part-1,part-2,part-3,part-4;
            COMBINATION        part-1 PATH-TYPE
                 WITH        part-2 RECORD-TYPE
                 WITH        part-3 VALUE-FOR ROLE-NAME
                 WITH        part-4 VALUE-FOR CONNECTIVITY;
            CONNECTIVITY       MANY part-1,
                                    part-2,
                                    part-3,
                                    part-4;
            CONNECTION-TYPE F1;
            STORED             part-1 1,
                               part-2 2,
                               part-3 3,
                               part-4 4;
            CODE       RELMEM 308;

STATEMENT member-part-1-stm;
            USED part-1 member-rel;
            FORM HAS part-2 AS part-3 WITH CONNECTIVITY part-4;

STATEMENT member-part-2-stm;
            USED part-2 member-rel;
            FORM IS part-3 OF part-1 WITH CONNECTIVITY part-4;

£-----------------------------------------------------------

RELATION Inverse-rel;
            PARTS              part-1,part-2;
            COMBINATION        part-1 PATH-TYPE
                   WITH        part-2 PATH-TYPE;
            CONNECTIVITY       ONE part-1,
                                   part-2;
            CONNECTION-TYPE S4;
            STORED             part-1 1,
                               part-2 2;
            CODE       RELINV 309;

STATEMENT Inverse-part-1-stm;
            USED part-1 Inverse-rel;
            FORM HAS part-2 AS INVERSE;

STATEMENT Inverse-part-2-stm;
            USED part-2 Inverse-rel;
            FORM INVERSE OF part-1;

£-----------------------------------------------------------
```

```
RELATION includes-rel;
            PARTS              part-1,part-2;
            COMBINATION        part-1 SCHEMA
                    WITH       part-2 PATH-TYPE;
            CONNECTIVITY       ONE  part-1,
                               MANY part-2;
            CONNECTION-TYPE    S2;
            STORED             part-1 1,
                               part-2 2;
            CODE       RELINC 310;

STATEMENT includes-part-1-stm;
            USED part-1 includes-rel;
            FORM INCLUDES part-2;

STATEMENT includes-part-2-stm;
            USED part-2 includes-rel;
            FORM IS CONTAINED IN part-1;

£-------------------------------------------------------------

RELATION works-on-rel;
            PARTS              part-1,part-2;
            COMBINATION        part-1 ACCESS-SCHEMA
                    WITH       part-2 MODULE;
            CONNECTIVITY       ONE  part-1
                               MANY part-2;
            CONNECTION-TYPE    S2;
            STORED             part-1 1,
                               part-2 2;
            CODE       RELWOR 311;

STATEMENT works-part-1-stm;
            USED part-1 works-on-rel;
            FORM IS MANIPULATED BY part-2;

STATEMENT works-part-2-stm;
            USED part-2 works-on-rel;
            FORM WORKS ON part-1;

£££££££££££££££££££££££££££££££££££££££££££££££££££££
£ VALUE-RANGE VALUE-RANGE VALUE-RANGE VALUE-RANGE          £
£££££££££££££££££££££££££££££££££££££££££££££££££££££

VALUE-RANGE ROLE-NAME;
            VALUES ORIGIN,TARGET;

VALUE-RANGE CONNECTIVITY;
            VALUES OM,NM,OO,NO;

VALUE-RANGE DIRECTION;
            VALUES ASCENDING,DESCENDING;

VALUE-RANGE SEQ-NUM;
            VALUES INTEGER 0 THRU 20;
```

DESCRIPTION OF THE ACCESS SCHEMA OF THE DATABASE WORKBENCH

```
*****************************************************************
*                                                               *
*     DESCRIPTION OF THE ACCESS SCHEMA OF THE DATABASE          *
*                        WORKBENCH                              *
*                                                               *
*****************************************************************
```

1. Record Types of the Access Schema

```
Record type SCHEMA:
                    SCH_TYPE : char(3)
                    SCH_NAME : char(12)
                    SCH_CODE : num(3)

Record type MODULE:
                    MD_TYPE : char(3)
                    MD_NAME : char(12)
                    MD_CODE : num(3)

Record type FILE:
                    F_TYPE : char(3)
                    F_NAME : char(12)
                    F_CODE : num(3)

Record type RECORD_TYPE:
                    RT_TYPE : char(3)
                    RT_NAME : char(12)
                    RT_CODE : num(3)

Record type PATH_TYPE:
                    AP_NAME : char(12)
                    AP_INVERSE : num(1)
                    AP_CODE : num(3)

Record type ITEM:
                    IT_NAME : char(12)
                    IT_CODE : num(3)
                    IT_SEQ  : num(3)
                    OPTIONAL: char(1)
                    REPETITIVITY : num(3)
                    LENGTH  : num(3)
                    TYPE    : char(3)
                    DECIM   : num(3)

Record type IKO:
                    IKO_CODE : num(3)
                    I-K-O    : char(3)
                    NUM_GLOBAL : num(3)
                    GLOBAL_ORDER : char(3)
                    SEG_GLOBAL : num(3)
                    SIMPLE_ORDER : char(3)
                    DUPLICATES : char(1)
                    REF_SET  : char(3)
```

```
Record type MEMBER:
                    ROLE : char(6)
                    MIN_MAX : char(3)

Record type COMPONENT:
                    DIRECTION : char(3)
                    SEQ_NUM   : num(3)

Record type IN:
                    /
```

2. Access Path Types of the Access Schema

```
Path type FILE_OF
        from one SCHEMA         optional
        to many FILE            optional
        inverse FILE_OF_I.

Path type FROM
        from one SCHEMA         optional
        to many RECORD_TYPE     optional
        inverse FROM_I.

Path type R_I
        from one RECORD_TYPE    optional
        to many IN              mandatory
        inverse R_I_I.

Path type I_F
        from one FILE           optional
        to many IN              mandatory
        inverse I_F_I.

Path type CONTAINS
        from one RECORD_TYPE    optional
        from one ITEM           optional
        to many ITEM            mandatory
        inverse CONTAINS_I.

Path type COUNTER
        from one ITEM           optional
        to many ITEM            optional
        inverse COUNTER_I.

Path type CONCERNS
        from one RECORD_TYPE    optional
        to many IKO             mandatory
        inverse CONCERNS_I.
```

```
Path type IKO_C
        from one IKO            optional
        to many COMPONENT       mandatory
        inverse IKO_C_I.

Path type C_I_P
        from one ITEM           optional
        from one PATH_TYPE      optional
        to many COMPONENT       mandatory
        inverse C_I_P_I.

Path type R_M
        from one RECORD_TYPE    optional
        to many MEMBER          mandatory
        inverse R_M_I.

Path type M_P
        from one PATH_TYPE      optional
        to many MEMBER          mandatory
        inverse M_P_I.

Path type INVERSE
        from one PATH_TYPE      optional
        to one PATH_TYPE        optional
        inverse INVERSE_I.

Path type INCLUDES
        from one SCHEMA         optional
        to many PATH_TYPE       mandatory
        inverse INCLUDES_I.

Path type WORKS_ON
        from one SCHEMA         optional
        to many MODULE          optional
        inverse WORKS_I.
```

```
**********************************
*                                *
*      LDF DEFINITION OF ADL     *
*                                *
**********************************
```

```
        ********************************
        *                              *
        *     LDF DEFINITION OF ADL    *
        *                              *
        ********************************

(* debut ldf *)
definition de adl;
<point_d_entree> ::= "algorithm" <name>$<intern_declaration_part>
    $<statement_part>^"."$ ;
<intern_declaration_part> ::= <type_declaration_part> $
    <variable_declaration_part>$ ;
SC<type_declaration_part> ::= <notype>| <type_declar> ;
<notype> ::= ;
<type_declar> ::= "type" <type_declarations> $ ;
L<type_declarations> ::= <type_declaration>+^";"$ ;
<type_declaration> ::= <name> "=" <variable_type> ;
SC<variable_declaration_part> ::= <novar> |<var_declar> ;
<novar> ::= ;
<var_declar> ::= "var" <variable_declarations> ;
L<variable_declarations> ::= <variable_declaration>+^";"$;
<variable_declaration> ::= <names> ":" <variable_type>;
L<names> ::= <name>+^"," ;
SC<variable_type> ::= <simple> | <structured> |<name>;
ALT<simple> ::= <boolean> | <str> | <real> | <integer> | <numeric>;
<boolean> ::= "boolean" ;
<str> ::= "string("^<unsigned_integer>^")" ;
<real> ::= "real" ;
<integer> ::= "integer" ;
<numeric> ::= "numeric("^<unsigned_integer>^","
    ^<unsigned_integer>^")" ;
ALT<structured> ::= <group> | <array> | <items_of> | <ref_name>
    | <ref_record> ;
<group> ::= "group"£ <field_list> $"end" ;
L<field_list> ::= <variable_declaration>+";"$ ;
<array> ::= "array ["^<index_lst>^"] of" <component_type> ;
SC<index_lst> ::= <index3> | <index2> | <index1> ;
<index3> ::= <unsigned_integer>^","^<unsigned_integer>^","
    ^<unsigned_integer> ;
<index2> ::= <unsigned_integer>^","^<unsigned_integer> ;
<index1> ::= <unsigned_integer> ;
ALT<component_type> ::= <simple> | <items_of> | <ref_name>
    | <ref_record> | <name> ;
<items_of> ::= "items_of" <name> ;
<ref_name> ::= "ref of" <name> ;
<ref_record> ::= "ref of RECORD" ;
<statement_part> ::= "begin" <statements> $"end" ;
L<statements> ::= <statement>+^";"$ ;
ALT<statement> ::= <ass_st> | <for_st> | <next_st> | <next_name_st>
    | <exit_st> | <exit_name_st> | <while_st> | <call_st> |
    <return_st> | <db_mod> | <if_st> | <if_else_st> | <dummy> ;
<dummy> ::= ;
<ass_st> ::= <variable> ":=" <assign_expression> ;
<for_st> ::= "for" <variable> ":=" <collection_expression> "do"£
    <statements> $"endfor" ;
<next_st> ::= "next" ;
<next_name_st> ::= "next" <name> ;
<exit_st> ::= "exit" ;
<exit_name_st> ::= "exit" <name> ;
```

```
<while_st> ::= "while" <general_expression> "do" £<statements> $
    "endwhile" ;
<if_st> ::= "if" <general_expression> £"then" <statements>$
    "endif" ;
<if_else_st> ::= "if" <general_expression> £"then" <statements>
    £"else" <statements> $ "endif" ;
<return_st> ::= "return"<param_eff> ;
<call_st> ::= <proc_name> <param_eff> ;
SC<param_eff> ::= <ss_param> | <param> ;
<ss_param> ::= ;
<param> ::= ^"("^<param_list>^")" ;
L<param_list> ::= <variable>+^"," ;
ALT<db_mod> ::= <modif> | <creat> | <del> ;
<creat> ::= "create" <name> ":=" <name>^<creat_conds>^ ;
SC<creat_conds> ::= <creat_cond> | <parenth_conditions> ;
<parenth_conditions> ::= ^"("^<conditions>^")"^ ;
L<conditions> ::= <creat_cond>+"and" ;
ALT<creat_cond> ::= <attach_cond> | <item_cond> ;
<modif> ::= "modify" <name>^<modif_conds> ;
ALT<modif_conds> ::= <attach_cond> | <items_conds> | <detach_cond>;
SC<items_conds> ::= <item_cond> | <parenth_conds> ;
<parenth_conds> ::= ^"("^<conds>^")"^ ;
L<conds> ::= <item_cond>+"and" ;
<item_cond> ::= ^"("^<qual>^":"^<name>^"="^<arithmetic_expression>
    ^")"^ ;
<attach_cond> ::= ^"("^<qual>^":"^<name>^")"^ ;
<detach_cond> ::= ^"("^<qual>^":"^<unsigned_integer><name>^")"^ ;
<del> ::= "delete" <name> ;
ALT<assign_expression> ::= <general_expression>
    | <arithmetic_expression> ;
SC<general_expression> ::= <general_factor> | <or_expression> ;
<or_expression> ::= <general_factor> "or" <general_expression> ;
SC<general_factor> ::= <general_term> | <and_expression> ;
<and_expression> ::= <general_term> "and" <general_factor> ;
SC<general_term> ::= <not_general_primary> | <general_primary> ;
<not_general_primary> ::= "not" <general_primary> ;
ALT<general_primary> ::= <test_expression> | <gen_expr_parenth> ;
<gen_expr_parenth> ::= ^"("^<general_expression>^")"^ ;
<test_expression> ::= <arithmetic_expression>^<test_operator>^
    <arithmetic_expression> ;
ALT<arithmetic_expression> ::= <add_arith> | <add> | <factor> ;
<add_arith> ::= <factor>^<adding>^<arithmetic_expression> ;
<add> ::= <adding>^<arithmetic_expression> ;
SC<factor> ::= <multipl_factor> | <term> ;
<multipl_factor> ::= <term>^<multiplying>^<factor> ;
SC<term> ::= <exp_term> | <primary> ;
<exp_term> ::= <primary>^<exp>^<primary> ;
<exp> ::= "^" ;
ALT<primary> ::= <string> | <unsigned_integer> | <unsigned_real>
    | <arith_expr_parenth> | <DB_object_set> | <variable>
    | <call_st> | <nullref> | <logical_value> ;
<nullref> ::= "()" ;
<arith_expr_parenth> ::= ^"("^<arithmetic_expression>^")"^ ;
SC<adding> ::= <plus> | <minus> ;
<plus> ::= "+" ;
<minus> ::= "-" ;
SC<multiplying> ::= <multipl> | <divided> ;
<multipl> ::= "*" ;
<divided> ::= "/" ;
```

```
ALT<collection_expression> ::= <range> | <DB_object_set> ;
<range> ::= <range_expr>^".."^<range_expr> ;
SC<range_expr> ::= <unsigned_integer> | <name> ;
<DB_object_set> ::= <name>^<predicate> ;
SC<predicate> ::= <and_ccterm> | <coll_cond_term> ;
<and_ccterm> ::= <coll_cond_term>"and"<predicate> ;
SC<coll_cond_term> ::= <not_ccprimary> | <coll_cond_primary> ;
<not_ccprimary> ::= "not" <coll_cond_primary> ;
ALT<coll_cond_primary> ::= <relation_condition> | <nopredicate> |
    <predicate_parenth> ;
<nopredicate> ::= "()" ;
<predicate_parenth> ::= ^"("^<predicate>^")"^ ;
<relation_condition> ::= ^"("^<qual>^":"^<prim_rel_cond>^")"^ ;
SC<prim_rel_cond> ::= <name> | <belonging_cond> ;
SC<qual> ::= <name> | <no_name> ;
<no_name> ::= ;
<belonging_cond> ::= <name>^<test_operator>
    ^<arithmetic_expression> ;
SC<unsigned_real> ::= <dec_exp_nb> | <decimal_number> ;
<dec_exp_nb> ::= <decimal_number>^"e"^<sign>^<unsigned_integer> ;
<decimal_number> ::= <unsigned_integer>^"."^<unsigned_integer> ;
SC<sign> ::= <plus> | <minus> | <no_sign> ;
<no_sign> ::= ;
ALT<variable> ::= <class_var> | <classical_var_items>
    | <var_items_file> | <var_items_type> ;
SC<class_var> ::= <hierar_variable> | <non_hierar_variable> ;
SC<non_hierar_variable> ::= <name> | <subscripted_variable> ;
<subscripted_variable> ::= <name>^"["^<subscript_list>^"]"^ ;
SC<subscript_list> ::= <subscr1> | <subscr2> | <subscr3> ;
<subscr1> ::= <arithmetic_expression> ;
<subscr2> ::= <arithmetic_expression>^","^<arithmetic_expression> ;
<subscr3> ::= <arithmetic_expression>^","^<arithmetic_expression>
    ^","^ <arithmetic_expression> ;
<hierar_variable> ::= <non_hierar_variable>^"."^<class_var> ;
<classical_var_items> ::= "("^<name>^")."^<class_var> ;
<var_items_file> ::= "("^<name>^").FILE" ;
<var_items_type> ::= "("^<name>^").TYPE" ;
SC<logical_value> ::= <true> | <false> ;
<true> ::= "true";
<false> ::= "false" ;
SC<test_operator> ::= <l_t> | <g_t> | <l_t_eq> | <g_t_eq>
    | <not_eq> | <eq> ;
<l_t> ::= "<" ;
<g_t> ::= ">" ;
<l_t_eq> ::= "<=" ;
<g_t_eq> ::= ">=" ;
<not_eq> ::= "<>" ;
<eq> ::= "=" ;
GEN <string> ::= "'[^\'\n]*'" ;
GEN <name> ::= "[A-Z][A-Z0-9_]*" ;
GEN <unsigned_integer> ::= "[0-9]+" ;
GEN <proc_name> ::= "[A-Z]([A-Z0-9_]*)""!""" .
```

```
*****************************************
*                                       *
*    LEX SPECIFICATIONS GENERATED       *
*          BY THE TRANSFORMATOR         *
*                                       *
*****************************************
```

```
*************************************
*                                   *
*   LEX SPECIFICATIONS GENERATED    *
*        BY THE TRANSFORMATOR       *
*                                   *
*************************************

%{
 int numlig;
%}
%%
"algorithm"                 return (T1 );
"."                         return (T2 );
"type"                      return (T3 );
";"                         return (T4 );
"="                         return (T5 );
"var"                       return (T6 );
":"                         return (T7 );
","                         return (T8 );
"boolean"                   return (T9 );
"string("                   return (T10 );
")"                         return (T11 );
"real"                      return (T12 );
"integer"                   return (T13 );
"numeric("                  return (T14 );
"group"                     return (T15 );
"end"                       return (T16 );
"array"[ \t\n]+"["                { rechcr(yytext,yyleng);
                            return (T17 );
                  };
"]"[ \t\n]+"of"                   { rechcr(yytext,yyleng);
                            return (T18 );
                  };
"items_of"                  return (T19 );
"ref"[ \t\n]+"of"                 { rechcr(yytext,yyleng);
                            return (T20 );
                  };
"ref"[ \t\n]+"of"[ \t\n]+"RECORD"    { rechcr(yytext,yyleng);
                            return (T21 );
                  };
"begin"                     return (T22 );
":="                        return (T23 );
"for"                       return (T24 );
"do"                        return (T25 );
"endfor"                    return (T26 );
"next"                      return (T27 );
"exit"                      return (T28 );
"while"                     return (T29 );
"endwnile"                  return (T30 );
"if"                        return (T31 );
"then"                      return (T32 );
"endif"                     return (T33 );
"else"                      return (T34 );
"return"                    return (T35 );
"("                         return (T36 );
"create"                    return (T37 );
"and"                       return (T38 );
```

```
'modify"                    return (T39 );
'delete"                    return (T40 );
'or"                        return (T41 );
'not"                       return (T42 );
"^"                         return (T43 );
"()"                        return (T44 );
"+"                         return (T45 );
"-"                         return (T46 );
"*"                         return (T47 );
"/"                         return (T48 );
".."                        return (T49 );
"e"                         return (T50 );
"["                         return (T51 );
"]"                         return (T52 );
").."                       return (T53 );
").FILE"                    return (T54 );
").TYPE"                    return (T55 );
"true"                      return (T56 );
"false"                     return (T57 );
"<"                         return (T58 );
">"                         return (T59 );
"<="                        return (T60 );
">="                        return (T61 );
"<>"                        return (T62 );
'[^\'\n]*'                  return (T63 );
[A-Z][A-Z0-9_]*             return (T64 );
[0-9]+                      return (T65 );
[A-Z]([A-Z0-9_]*)"!"        return (T66 );
[ \t]                      ;
[\n]                      { numlig +=1;};
.                  ;

%%
 yywrap() {
return (1) ;
      }
```

```
**************************************
*                                    *
*     YACC SPECIFICATIONS GENERATED  *
*          BY THE TRANSFORMATOR      *
*                                    *
**************************************
```

```
*****************************************
*                                       *
*     YACC SPECIFICATIONS GENERATED     *
*          BY THE TRANSFORMATOR         *
*                                       *
*****************************************

%{
£ include <stdio.h>
£ include <ctype.h>
int numlig ;
int errorenc;
int erriig;
int cpterr;
%}
%token  T1
%token  T2
%token  T3
%token  T4
%token  T5
%token  T6
%token  T7
%token  T8
%token  T9
%token  T10
%token  T11
%token  T12
%token  T13
%token  T14
%token  T15
%token  T16
%token  T17
%token  T18
%token  T19
%token  T20
%token  T21
%token  T22
%token  T23
%token  T24
%token  T25
%token  T26
%token  T27
%token  T28
%token  T29
%token  T30
%token  T31
%token  T32
%token  T33
%token  T34
%token  T35
%token  T36
%token  T37
%token  T38
%token  T39
%token  T40
%token  T41
%token  T42
%token  T43
%token  T44
```

```
%token  T45
%token  T46
%token  T47
%token  T48
%token  T49
%token  T50
%token  T51
%token  T52
%token  T53
%token  T54
%token  T55
%token  T56
%token  T57
%token  T58
%token  T59
%token  T60
%token  T61
%token  T62
%token  T63
%token  T64
%token  T65
%token  T66
%start XXXXXXXXXX
%%
XXXXXXXXXX   : point_d_entree
             { $$=$1;
               somm($$);
                return (1); }
              ;

point_d_entree : T1 name intern_declaration_part statement_part T2
               {$$=nter (9,$2,$3,$4);}
             ;

intern_declaration_part : type_declaration_part
                          variable_declaration_part
               {$$=nbin (23,$1,$2);}
             ;

type_declaration_part        : notype
               { $$= $1;}
             | type_declar
               { $$= $1;}
             ;

notype       :
               {$$=nzer(110 );}
             ;

type_declar : T3 type_declarations
               {$$=nun (48,hliste(1 ,$2) );}
             ;

type_declarations   : type_declarations T4 type_declaration
               {$$=clien($1,$3);}
             | type_declaration
               {$$=$1;}
             ;
```

```
type_declaration     : name T5 variable_type
                       {$$=nbin (24,$1,$3);}
                     ;

variable_declaration_part    : novar
                       { $$= $1;}
                     | var_declar
                       { $$= $1;}
                     ;

novar                :
                       {$$=nzer(111 );}
                     ;

var_declar           : T6 variable_declarations
                       {$$=nun (50,hliste(2 ,$2) );}
                     ;

variable_declarations : variable_declarations T4
                                variable_declaration
                       {$$=clien($1,$3);}
                     | variable_declaration
                       {$$=$1;}
                     ;

variable_declaration          : names T7 variable_type
                       {$$=nbin (25,nliste(3 ,$1) ,$3);}
                     ;

names                : names T8 name
                       {$$=clien($1,$3);}
                     | name
                       {$$=$1;}
                     ;

variable_type              : simple
                       { $$= $1;}
                     | structured
                       { $$= $1;}
                     | name
                       { $$= $1;}
                     ;

simple               : boolean
                       { $$= nun (52,$1);}
                     | str
                       { $$= nun (52,$1);}
                     | real
                       { $$= nun (52,$1);}
                     | integer
                       { $$= nun (52,$1);}
                     | numeric
                       { $$= nun (52,$1);}
                     ;

boolean              : T9
                       {$$=nzer(112 );}
                     ;
```

```
str          : T10 unsigned_integer T11
               {$$=nun (53,$2);}
             ;

real         : T12
               {$$=nzer(113 );}
             ;

integer      : T13
               {$$=nzer(114 );}
             ;

numeric      : T14 unsigned_integer T8 unsigned_integer T11
               {$$=nbin (26,$2,$4);}
             ;

structured   : group
               { $$= nun (54,$1);}
             | array
               { $$= nun (54,$1);}
             | items_of
               { $$= nun (54,$1);}
             | ref_name
               { $$= nun (54,$1);}
             | ref_record
               { $$= nun (54,$1);}
             ;

group        : T15 field_list T16
               {$$=nun (55,hliste(4 ,$2) );}
             ;

field_list   : field_list T4 variable_declaration
               {$$=clien($1,$3);}
             | variable_declaration
               {$$=$1;}
             ;

array        : T17 index_lst T18 component_type
               {$$=nbin (27,$2,$4);}
             ;

index_lst    : index3
               { $$= $1;}
             | index2
               { $$= $1;}
             | index1
               { $$= $1;}
             ;

index3 : unsigned_integer T8 unsigned_integer T8 unsigned_integer
               {$$=nter (10,$1,$3,$5);}
             ;

index2       : unsigned_integer T8 unsigned_integer
               {$$=nbin (28,$1,$3);}
             ;
```

```
index1          : unsigned_integer
                  {$$=nun (57,$1);}
                ;

component_type          : simple
                  { $$= nun (58,$1);}
                | items_of
                  { $$= nun (58,$1);}
                | ref_name
                  { $$= nun (58,$1);}
                | ref_record
                  { $$= nun (58,$1);}
                | name
                  { $$= nun (58,$1);}
                ;

items_of        : T19 name
                  {$$=nun (59,$2);}
                ;

ref_name        : T20 name
                  {$$=nun (60,$2);}
                ;

ref_record      : T21
                  {$$=nzer(115 );}
                ;

statement_part          : T22 statements T16
                  {$$=nun (61,hliste(5 ,$2) );}
                ;

statements      : statements T4 statement
                  {$$=clien($1,$3);}
                | statement
                  {$$=$1;}
                ;

statement       : ass_st
                  { $$= nun (62,$1);}
                | for_st
                  { $$= nun (62,$1);}
                | next_st
                  { $$= nun (62,$1);}
                | next_name_st
                  { $$= nun (62,$1);}
                | exit_st
                  { $$= nun (62,$1);}
                | exit_name_st
                  { $$= nun (62,$1);}
                | while_st
                  { $$= nun (62,$1);}
                | call_st
                  { $$= nun (62,$1);}
                | return_st
                  { $$= nun (62,$1);}
```

```
                | db_mod
                  { $$= nun (62,$1);}
                | if_st
                  { $$= nun (62,$1);}
                | if_else_st
                  { $$= nun (62,$1);}
                | dummy
                  { $$= nun (62,$1);}
                ;

dummy           :
                  {$$=nzer(116 );}
                ;

ass_st          : variable T23 assign_expression
                  {$$=nbin (29,$1,$3);}
                ;

for_st : T24 variable T23 collection_expression T25 statements T26
                  {$$=nter (11,$2,$4,hliste(5 ,$6) );}
                ;

next_st         : T27
                  {$$=nzer(117 );}
                ;

next_name_st            : T27 name
                  {$$=nun (63,$2);}
                ;

exit_st         : T28
                  {$$=nzer(118 );}
                ;

exit_name_st            : T28 name
                  {$$=nun (64,$2);}
                ;

while_st        : T29 general_expression T25 statements T30
                  {$$=nbin (30,$2,hliste(5 ,$4) );}
                ;

if_st           : T31 general_expression T32 statements T33
                  {$$=nbin (31,$2,hliste(5 ,$4) );}
                ;

if_else_st      : T31 general_expression T32 statements T34
                        statements T33
                  {$$=nter (12,$2,hliste(5 ,$4) ,hliste(5 ,$6) );}
                ;

return_st       : T35 param_eff
                  {$$=nun (65,$2);}
                ;

call_st         : proc_name param_eff
                  {$$=nbin (32,$1,$2);}
                ;
```

```
param_eff    : ss_param
               { $$= $1;}
             | param
               { $$= $1;}
             ;

ss_param     :
               {$$=nzer(119 );}
             ;

param        : T36 param_list T11
               {$$=nun (67,hliste(6 ,$2) );}
             ;

param_list   : param_list T8 variable
               {$$=clien($1,$3);}
             | variable
               {$$=$1;}
             ;

db_mod       : modif
               { $$= nun (68,$1);}
             | creat
               { $$= nun (68,$1);}
             | del
               { $$= nun (68,$1);}
             ;

creat        : T37 name T23 name creat_conds
               {$$=nter (13,$2,$4,$5);}
             ;

creat_conds  : creat_cond
               { $$= $1;}
             | parenth_conditions
               { $$= $1;}
             ;

parenth_conditions  : T36 conditions T11
               {$$=nun (70,hliste(7 ,$2) );}
             ;

conditions   : conditions T38 creat_cond
               {$$=clien($1,$3);}
             | creat_cond
               {$$=$1;}
             ;

creat_cond   : attach_cond
               { $$= nun (71,$1);}
             | item_cond
               { $$= nun (71,$1);}
             ;

modif        : T39 name modif_conds
               {$$=nbin (33,$2,$3);}
             ;
```

```
modif_conds : attach_cond
              { $$= nun (72,$1);}
            | items_conds
              { $$= nun (72,$1);}
            | detach_cond
              { $$= nun (72,$1);}
            ;

items_conds : item_cond
              { $$= $1;}
            | parenth_conds
              { $$= $1;}
            ;

parenth_conds       : T36 conds T11
              {$$=nun (74,hliste(8 ,$2) );}
            ;

conds       : conds T38 item_cond
              {$$=clien($1,$3);}
            | item_cond
              {$$=$1;}
            ;

item_cond   : T36 qual T7 name T5 arithmetic_expression T11
              {$$=nter (14,$2,$4,$6);}
            ;

attach_cond : T36 qual T7 name T11
              {$$=nbin (34,$2,$4);}
            ;

detach_cond : T36 qual T7 unsigned_integer name T11
              {$$=nter (15,$2,$4,$5);}
            ;

del         : T40 name
              {$$=nun (75,$2);}
            ;

assign_expression   : general_expression
              { $$= nun (76,$1);}
            | arithmetic_expression
              { $$= nun (76,$1);}
            ;

general_expression  : general_factor
              { $$= $1;}
            | or_expression
              { $$= $1;}
            ;

or_expression       : general_factor T41 general_expression
              {$$=nbin (35,$1,$3);}
            ;
```

```
general_factor      : general_term
                      { $$= $1;}
                    | and_expression
                      { $$= $1;}
                    ;

and_expression      : general_term T38 general_factor
                      {$$=nbin (36,$1,$3);}
                    ;

general_term        : not_general_primary
                      { $$= $1;}
                    | general_primary
                      { $$= $1;}
                    ;

not_general_primary : T42 general_primary
                      {$$=nun (80,$2);}
                    ;

general_primary     : test_expression
                      { $$= nun (81,$1);}
                    | gen_expr_parenth
                      { $$= nun (81,$1);}
                    ;

gen_expr_parenth    : T36 general_expression T11
                      {$$=nun (82,$2);}
                    ;

test_expression : arithmetic_expression test_operator
                      arithmetic_expression
                      {$$=nter (16,$1,$2,$3);}
                    ;

arithmetic_expression       : add_arith
                      { $$= nun (83,$1);}
                    | add
                      { $$= nun (83,$1);}
                    | factor
                      { $$= nun (83,$1);}
                    ;

add_arith   : factor adding arithmetic_expression
                      {$$=nter (17,$1,$2,$3);}
                    ;

add         : adding arithmetic_expression
                      {$$=nbin (37,$1,$2);}
                    ;

factor      : multipl_factor
                      { $$= $1;}
                    | term
                      { $$= $1;}
                    ;

multipl_factor      : term multiplying factor
                      {$$=nter (18,$1,$2,$3);}
                    :
```

```
term            : exp_term
                  { $$= $1;}
                | primary
                  { $$= $1;}
                ;

exp_term        : primary exp primary
                  {$$=nter (19,$1,$2,$3);}
                ;

exp             : T43
                  {$$=nzer(120 );}
                ;

primary         : string
                  { $$= nun (86,$1);}
                | unsigned_integer
                  { $$= nun (86,$1);}
                | unsigned_real
                  { $$= nun (86,$1);}
                | arith_expr_parenth
                  { $$= nun (86,$1);}
                | DB_object_set
                  { $$= nun (86,$1);}
                | variable
                  { $$= nun (86,$1);}
                | call_st
                  { $$= nun (86,$1);}
                | nullref
                  { $$= nun (86,$1);}
                | logical_value
                  { $$= nun (86,$1);}
                ;

nullref         : T44
                  {$$=nzer(121 );}
                ;

arith_expr_parenth  : T36 arithmetic_expression T11
                  {$$=nun (87,$2);}
                ;

adding          : plus
                  { $$= $1;}
                | minus
                  { $$= $1;}
                ;

plus            : T45
                  {$$=nzer(122 );}
                ;

minus           : T46
                  {$$=nzer(123 );}
                ;
```

```
multiplying : multipl
              { $$= $1;}
            | divided
              { $$= $1;}
            ;

multipl     : T47
              {$$=nzer(124 );}
            ;

divided     : T48
              {$$=nzer(125 );}
            ;

collection_expression       : range
              { $$= nun (90,$1);}
            | DB_object_set
              { $$= nun (90,$1);}
            ;

range       : range_expr T49 range_expr
              {$$=nbin (38,$1,$3);}
            ;

range_expr  : unsigned_integer
              { $$= $1;}
            | name
              { $$= $1;}
            ;

DB_object_set       : name predicate
              {$$=nbin (39,$1,$2);}
            ;

predicate   : and_ccterm
              { $$= $1;}
            | coll_cond_term
              { $$= $1;}
            ;

and_ccterm  : coll_cond_term T38 predicate
              {$$=nbin (40,$1,$3);}
            ;

coll_cond_term      : not_ccprimary
              { $$= $1;}
            | coll_cond_primary
              { $$= $1;}
            ;

not_ccprimary       : T42 coll_cond_primary
              {$$=nun (94,$2);}
            ;
```

```
coll_cond_primary   : relation_condition
                      { $$= nun (95,$1);}
                    | nopredicate
                      { $$= nun (95,$1);}
                    | predicate_parenth
                      { $$= nun (95,$1);}
                    ;

nopredicate         : T44
                      {$$=nzer(126 );}
                    ;

predicate_parenth   : T36 predicate T11
                      {$$=nun (96,$2);}
                    ;

relation_condition  : T36 qual T7 prim_rel_cond T11
                      {$$=nbin (41,$2,$4);}
                    ;

prim_rel_cond       : name
                      { $$= $1;}
                    | belonging_cond
                      { $$= $1;}
                    ;

qual                : name
                      { $$= $1;}
                    | no_name
                      { $$= $1;}
                    ;

no_name             :
                      {$$=nzer(127 );}
                    ;

belonging_cond      : name test_operator aritnmetic_expression
                      {$$=nter (20,$1,$2,$3);}
                    ;

unsigned_real       : dec_exp_nb
                      { $$= $1;}
                    | decimal_number
                      { $$= $1;}
                    ;

dec_exp_nb  : decimal_number T50 sign unsigned_integer
              {$$=nter (21,$1,$3,$4);}
            ;

decimal_number      : unsigned_integer T2 unsigned_integer
                      {$$=nbin (42,$1,$3);}
                    ;
```

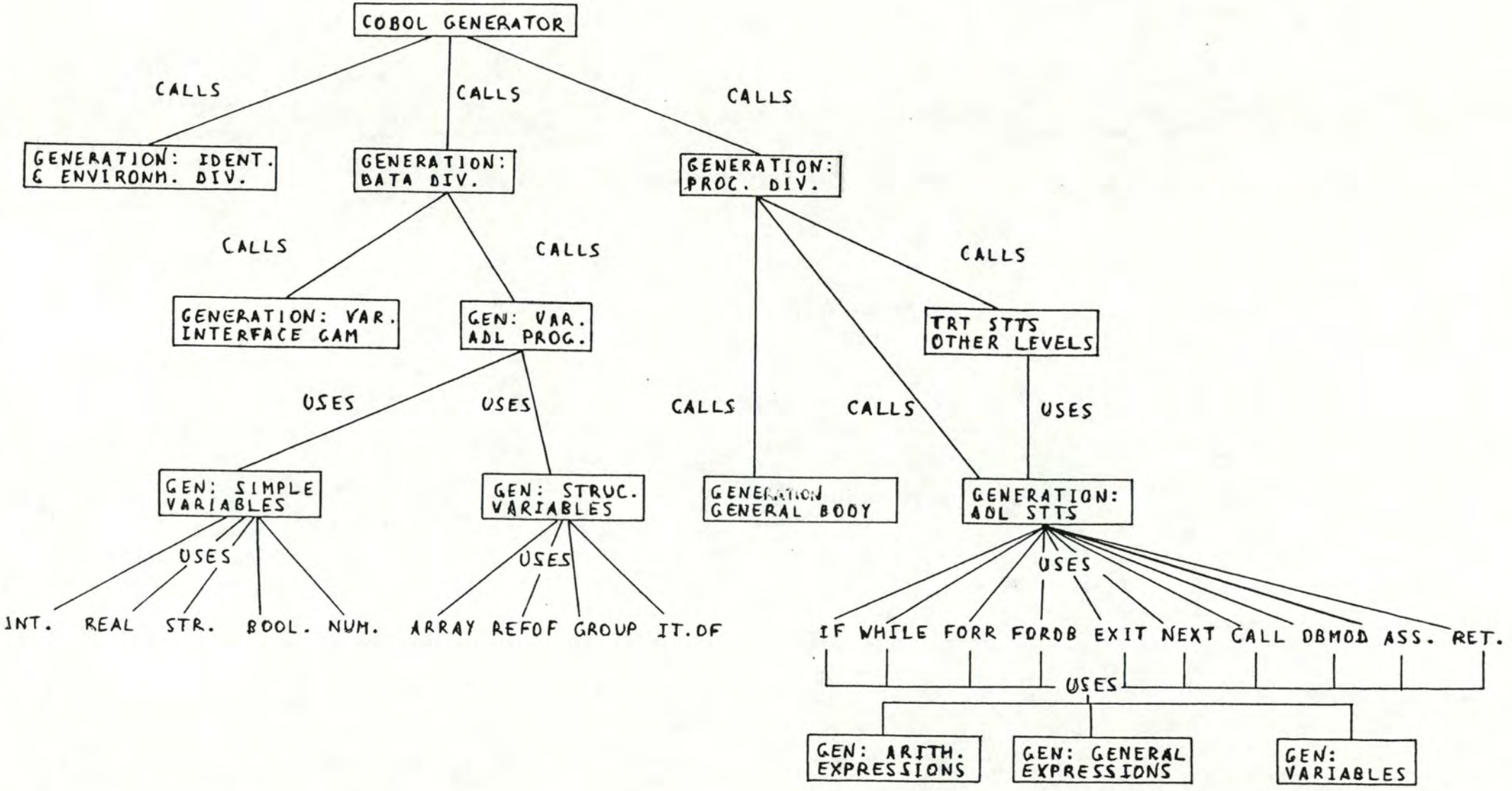
```
sign            : plus
                  { $$= $1;}
                | minus
                  { $$= $1;}
                | no_sign
                  { $$= $1;}
                ;

no_sign         :
                  {$$=nzer(128 );}
                ;

variable        : class_var
                  { $$= nun (101,$1);}
                | classical_var_items
                  { $$= nun (101,$1);}
                | var_items_file
                  { $$= nun (101,$1);}
                | var_items_type
                  { $$= nun (101,$1);}
                ;

class_var       : hierar_variable
                  { $$= $1;}
                | non_hierar_variable
                  { $$= $1;}
                ;

non_hierar_variable : name
                  { $$= $1;}
                | subscripted_variable
                  { $$= $1;}
                ;

subscripted_variable          : name T51 subscript_list T52
                  {$$=nbin (43,$1,$3);}
                ;

subscript_list        : subscr1
                  { $$= $1;}
                | subscr2
                  { $$= $1;}
                | subscr3
                  { $$= $1;}
                ;

subscr1         : arithmetic_expression
                  {$$=nun (105,$1);}
                ;

subscr2         : arithmetic_expression T8 arithmetic_expression
                  {$$=nbin (44,$1,$3);}
                ;
```

```
subscr3      : aritnmetic_expression T8 arithmetic_expression
                     T8 arithmetic_expression
               {$$=nter (22,$1,$3,$5);}
             ;

hierar_variable      : non_hierar_variable T2 class_var
               {$$=nbin (45,$1,$3);}
             ;

classical_var_items : T36 name T53 class_var
               {$$=nbin (46,$2,$4);}
             ;

var_items_file       : T36 name T54
               {$$=nun (106,$2);}
             ;

var_items_type       : T36 name T55
               {$$=nun (107,$2);}
             ;

logical_value        : true
               { $$= $1;}
             | false
               { $$= $1;}
             ;

true         : T56
               {$$=nzer(129 );}
             ;

false        : T57
               {$$=nzer(130 );}
             ;

test_operator        : l_t
               { $$= $1;}
             | g_t
               { $$= $1;}
             | l_t_eq
               { $$= $1;}
             | g_t_eq
               { $$= $1;}
             | not_eq
               { $$= $1;}
             | eq
               { $$= $1;}
             ;

l_t          : T58
               {$$=nzer(131 );}
             ;

g_t          : T59
               {$$=nzer(132 );}
             ;
```

```
l_t_eq          : T60
                  {$$=nzer(133 );}
                ;

g_t_eq          : T61
                {$$=nzer(134 );}
                ;

not_eq          : T62
                  {$$=nzer(135 );}
                ;

eq              : T5
                  {$$=nzer(136 );}
                ;

string          :  T63
                  {$$=constrgen(137,yytext,yyleng);}
                ;

name            :  T64
                  {$$=constrgen(138,yytext,yyleng);}
                ;

unsigned_integer    :  T65
                  {$$=constrgen(139,yytext,yyleng);}
                ;

proc_name       :  T66
                  {$$=constrgen(140,yytext,yyleng);}
                ;

%%
# include "lex.ad.c"
yyerror (s)
   char *s ;
  { int i;
    erreurc = 1;
    if (numlig == errlig)
            { cpterr +=1; }
      else
             { errlig =numlig;
               cpterr =1;};
               i = yytext[0];
    if ( i == 0)
      { fprintf (yyout,"erreur syntaxique a la ligne %d",numlig);
        fprintf (yyout,"...fin de texte inattendue\n");
        cpterr = 11 ;
        return (0);
      };
    fprintf (yyout,"erreur syntaxique a la ligne %d",numlig);
    fprintf (yyout,"...caracteres inattendus : ");
    for (i=0;i<yyleng;i++)
          {putc (yytext[i],yyout);};
    fprintf(yyout,"\n");
    if (cpterr >= 10)
       { fprintf (yyout,"trop d'erreurs a la ligne %d",numlig);
         fprintf (yyout,"\n AU REVOIR ! ! !\n");
       }
}
```

ARCHITECTURE OF THE COBOL GENERATOR

This part of the appendix describes the architecture of the Cobol generator by giving, first a schema of the generator and then the specification of each of its componants.


COBOL GENERATOR
CALLS
CALLS
CALLS
GENERATION: IDENT. & ENVIRONM. DIV.
GENERATION: DATA DIV.
GENERATION: PROC. DIV.
CALLS
CALLS
CALLS
GENERATION: VAR. INTERFACE GAM
GEN: VAR. ADL PROG.
TRT STTS OTHER LEVELS
USES
USES
CALLS
CALLS
USES
GEN: SIMPLE VARIABLES
GEN: STRUC. VARIABLES
GENERATION GENERAL BODY
GENERATION: ADL STTS
USES
USES
INT.
REAL
STR.
BOOL.
NUM.
ARRAY
REFOF
GROUP
IT.OF
IF
WHILE
FORR
FORDB
EXIT
NEXT
CALL
DBMOD
ASS.
RET.
USES
GEN: ARITH. EXPRESSIONS
GEN: GENERAL EXPRESSIONS
GEN: VARIABLES

Specification of each component of the generator.
****

COBOL GENERATOR.

Function: it generates the Cobol program corresponding to an ADL program stored in the system by means of its parse tree.

It calls the following components:
- GENERATION OF THE IDENTIFICATION & ENVIRONMENT DIVISIONS;
- GENERATION OF THE DATA DIVISION;
- GENERATION OF THE PROCEDURE DIVISION.

GENERATION OF THE IDENTIFICATION & ENVIRONMENT DIVISIONS.

Function: it generates the identification division and the environment division according to their generation principles (see chapter 7).

GENERATION OF THE DATA DIVISION.

Function: it generates the data division of the Cobol program.

It calls the following components:
- GENERATION: VARIABLES USED AS INTERFACE WITH THE GAM;
- GENERATION: VARIABLES DECLARED IN THE ADL PROGRAM.

GENERATION: VARIABLES USED AS INTERFACE WITH THE GAM.

Function: it generates the variables used in the interface with the GAM and the different codes of the database objects used in the program. This generation is performed according to the generation principles.

It uses the following component:
- GET INFO TABLES.

GENERATION: VARIABLES DECLARED IN THE ADL PROGRAM.

Function: it generates, for each declaration of variables found in the ADL program, the corresponding Cobol declaration.

It uses the following components:
- GENERATION: SIMPLE VARIABLES;
- GENERATION: STRUCTURED VARIABLES.

GENERATION: SIMPLE VARIABLES.

Function: it generates a Cobol declaration for a simple data type.

It uses the following components:
- INTEGER DECLARATION;
- REAL DECLARATION;
- STRING DECLARATION;
- BOOLEAN DECLARATION;
- NUMERIC DECLARATION.

INTEGER DECLARATION.

Function: according to the generation principles, it generates a Cobol declaration for a variable whose type is integer.

REAL DECLARATION.

Function: according to the generation principles, it generates a Cobol declaration for a variable whose type is real.

STRING DECLARATION.

Function: according to the generation principles, it generates a Cobol declaration for a variable whose type is string.

BOOLEAN DECLARATION.

Function: according to the generation principles, it generates a Cobol declaration for a variable whose type is boolean.

NUMERIC DECLARATION.

Function: according to the generation principles, it generates a Cobol declaration for a variable whose type is numeric.

GENERATION: STRUCTURED VARIABLES.

Function: it generates a Cobol declaration for a structured data type.

It uses the following components:
- ARRAY DECLARATION;
- REFOF DECLARATION;
- GROUP DECLARATION;
- ITEMS OF DECLARATION.

ARRAY DECLARATION.

Function: according to the generation principles, it generates a Cobol declaration for a variable that is an array.

REFOF DECLARATION.

Function: according to the generation principles, it generates a Cobol declaration for a variable whose type is ref of.

It uses the following component:
- GET INFO TABLES.

GROUP DECLARATION.

Function: according to the generation principles, it generates a Cobol declaration for a variable that is a group.

ITEMS OF DECLARATION.

Function: according to the generation principles, it generates a Cobol declaration for a variable whose type is items of.

It uses the following component:
- GET INFO TABLES.

GENERATION OF THE PROCEDURE DIVISION.

Function: it generates the procedure division; that is the Cobol statements corresponding to the ADL statements.

It calls the following components:
- GENERATION: GENERAL BODY;
- GENERATION: ADL STATEMENTS;
- TRT STTS OTHER LEVELS.

GENERATION: GENERAL BODY.

Function: it generates the general body of the procedure division according to the generation principles. The general body includes, among others, statements doing the opening and closing of the database.

GENERATION: ADL STATEMENTS.

Function: it generates the Cobol statements corresponding to the body of the ADL program.

It uses the following components:
- GENERATION: IF STATEMENT;
- GENERATION: WHILE STATEMENT;
- GENERATION: FORR STATEMENT;
- GENERATION: FORDB STATEMENT;
- GENERATION: EXIT STATEMENT;
- GENERATION: NEXT STATEMENT;
- GENERATION: CALL STATEMENT;
- GENERATION: DBMOD STATEMENTS;
- GENERATION: ASSIGN STATEMENT;
- GENERATION: RETURN STATEMENT.

GENERATION: IF STATEMENT.

Function: it generates the Cobol statements corresponding to an ADL if statement according to its generation principles.

It uses the following components:
- GENERATION: GENERAL EXPRESSIONS;
- GENERATION: VARIABLES.

GENERATION: WHILE STATEMENT.

Function: it generates the Cobol statements corresponding to an ADL while statement according to its generation principles.

It uses the following components:
- GENERATION: GENERAL EXPRESSIONS;
- GENERATION: VARIABLES.

GENERATION: FORR STATEMENT.

Function: it generates the Cobol statements corresponding to an ADL forr (for range) statement according to its generation principles.

It uses the following components:
- GENERATION: ARITHMETIC EXPRESSIONS;
- GENERATION: VARIABLES.

GENERATION: FORDB STATEMENT.

Function: it generates the Cobol statements corresponding to an ADL fordb (for database) statement according to its generation principles.

It uses the following components:
- GENERATION: ARITHMETIC EXPRESSIONS;
- GENERATION: VARIABLES.

GENERATION: EXIT STATEMENT.

Function: it generates the Cobol statements corresponding to an ADL exit statement according to its generation principles.

It uses the following component:
- GENERATION: VARIABLES.

GENERATION: NEXT STATEMENT.

Function: it generates the Cobol statements corresponding to an ADL next statement according to its generation principles.

It uses the following component:
- GENERATION: VARIABLES.

GENERATION: CALL STATEMENT.

Function: it generates the Cobol statements corresponding to an ADL call statement according to its generation principles.

It uses the following component:
- GENERATION: VARIABLES.

GENERATION: DBMOD STATEMENTS.

Function: it generates the Cobol statements corresponding to an ADL dbmod (database modification) statement according to its generation principles.

It uses the following components:
- GET INFO TABLES;
- GENERATION: GENERAL EXPRESSIONS;
- GENERATION: ARITHMETIC EXPRESSIONS;
- GENERATION: VARIABLES.

GENERATION: ASSIGN STATEMENT.

Function: it generates the Cobol statements corresponding to an ADL assignation statement according to its generation principles.

It uses the following components:
- GENERATION: GENERAL EXPRESSIONS;
- GENERATION: ARITHMETIC EXPRESSIONS;
- GENERATION: VARIABLES.

GENERATION: RETURN STATEMENT.

Function: it generates the Cobol statements corresponding to an ADL return statement according to its generation principles.

It uses the following component:
- GENERATION: VARIABLES.

TRT STATEMENTS OTHER LEVELS.

Function: it generates the Cobol statements corresponding to the ADL statements controled by control statements (if while for).

It uses the following component:
- GENERATION: ADL STATEMENTS.

GENERATION: GENERAL EXPRESSIONS.

Function: it generates the Cobol general expression corresponding to an ADL general expression, according to its generation principles.

It uses the following component:
- GENERATION: ARITHMETIC EXPRESSIONS;
- GENERATION: VARIABLES.

GENERATION: ARITHMETIC EXPRESSIONS.

Function: it generates the Cobol arithmetic expression corresponding to an ADL arithmetic expression, according to its generation principles.

It uses the following component:
- GENERATION: VARIABLES.

GENERATION: VARIABLES.

Function: it generates a Cobol variable corresponding to an ADL variable, according to its generation principles.

GET INFO TABLES.

Function: it accesses the information stored by the semantic analyser (for more details concerning the functions offered see chapter 6).

!!!! ALL COMPONENTS USE THE TREE ACCESS FUNCTIONS. !!!!

TREE ACCESS FUNCTIONS.

FATHER : access to the father of a node;
SON : access to the son of a node;
BROTHER: access to the brother of a node;
CODE : access to the field 'code' of a node;
VALUE : access to the field 'value' of a node.

Remark: the generation principles and, thus, the examples given in the appendix do not take into account the last version of the functions offered by the GAM (the appendix includes only the last version).

```
**********************************
*                                *
*      DECOMPILED ADL TEXT,      *
*     HIERARCHICAL TREE AND      *
*  COBOL PROGRAM GENERATED FROM  *
*    THE EXAMPLE OF CHAPTER 8    *
*                                *
**********************************
```

```
        ************************************
        *                                  *
        *    DECOMPILED ADL TEXT FROM      *
        *    THE EXAMPLE OF CHAPTER 8      *
        *                                  *
        ************************************

(*debut adl *)
algorithm FUNC

var CNBCUST : integer;
    CUST : ref of CUSTOMER;
    ORD : ref of ORDER;
    OL : ref of ORDER_LINE;
    PROD : ref of PRODUCT

begin CNBCUST := 0;
      for CUST := CUSTOMER () do
          for ORD := ORDER( C_OR: CUST) do
              if ( ORD). DATE = 310835
                  then for OL := ORDER_LINE( OR_OL: ORD) do
                               for PROD := PRODUCT( OL_P: OL) do
                                   if ( PROD). NBPRO = 5159
                                       then CNBCUST := CNBCUST + 1;
                                            next CUST;

                                   endif
                               endfor
                          endfor
              endif
          endfor
      endfor
end.
```

```
        ****************************************
        *                                      *
        *     HIERARCHICAL TREE OF THE         *
        *       EXAMPLE OF CHAPTER 8           *
        *                                      *
        ****************************************

!annot
!!( formalisme : -1 ) adl
!! ( point_d_entree: 9 )
!!! ( name: -138 ) VALEUR :FUNC
!!! ( intern_declaration_part: 23 )
!!!! ( notype: 110 )
!!!! ( var_declar: 50 )
!!!!! ( variable_declarations: 2 )
!!!!!! ( variable_declaration: 25 )
!!!!!!! ( names: 3 )
!!!!!!!! ( name: -138 ) VALEUR :CNBCUST
!!!!!!! ( simple: 52 )
!!!!!!!! ( integer: 114 )
!!!!!! ( variable_declaration: 25 )
!!!!!!! ( names: 3 )
!!!!!!!! ( name: -138 ) VALEUR :CUST
!!!!!!! ( structured: 54 )
!!!!!!!! ( ref_name: 60 )
!!!!!!!!! ( name: -138 ) VALEUR :CUSTOMER
!!!!!! ( variable_declaration: 25 )
!!!!!!! ( names: 3 )
!!!!!!!! ( name: -138 ) VALEUR :ORD
!!!!!!! ( structured: 54 )
!!!!!!!! ( ref_name: 60 )
!!!!!!!!! ( name: -138 ) VALEUR :ORDER
!!!!!! ( variable_declaration: 25 )
!!!!!!! ( names: 3 )
!!!!!!!! ( name: -138 ) VALEUR :OL
!!!!!!! ( structured: 54 )
!!!!!!!! ( ref_name: 60 )
!!!!!!!!! ( name: -138 ) VALEUR :ORDER_LINE
!!!!!! ( variable_declaration: 25 )
!!!!!!! ( names: 3 )
!!!!!!!! ( name: -138 ) VALEUR :PROD
!!!!!!! ( structured: 54 )
!!!!!!!! ( ref_name: 60 )
!!!!!!!!! ( name: -138 ) VALEUR :PRODUCT
!!! ( statement_part: 61 )
!!!! ( statements: 5 )
!!!!! ( statement: 62 )
!!!!!! ( ass_st: 29 )
!!!!!!! ( variable: 101 )
!!!!!!!! ( name: -138 ) VALEUR :CNBCUST
!!!!!!! ( assign_expression: 76 )
!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!! ( primary: 86 )
!!!!!!!!!! ( unsigned_integer: -139 ) VALEUR :0
!!!!! ( statement: 62 )
!!!!!! ( for_st: 11 )
!!!!!!! ( variable: 101 )
!!!!!!!! ( name: -138 ) VALEUR :CUST
```

```
!!!!!!! ( collection_expression: 90 )
!!!!!!!! ( DB_object_set: 39 )
!!!!!!!!! ( name: -138 ) VALEUR :CUSTOMER
!!!!!!!!! ( coll_cond_primary: 95 )
!!!!!!!!!! ( nopredicate: 126 )
!!!!!!! ( statements: 5 )
!!!!!!!! ( statement: 62 )
!!!!!!!!! ( for_st: 11 )
!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!! ( name: -138 ) VALEUR :ORD
!!!!!!!!!! ( collection_expression: 90 )
!!!!!!!!!!! ( DB_object_set: 39 )
!!!!!!!!!!!! ( name: -138 ) VALEUR :ORDER
!!!!!!!!!!!! ( coll_cond_primary: 95 )
!!!!!!!!!!!!! ( relation_condition: 41 )
!!!!!!!!!!!!!! ( name: -138 ) VALEUR :C_OR
!!!!!!!!!!!!!! ( name: -138 ) VALEUR :CUST
!!!!!!!!!! ( statements: 5 )
!!!!!!!!!!! ( statement: 62 )
!!!!!!!!!!!! ( if_st: 31 )
!!!!!!!!!!!!! ( general_primary: 81 )
!!!!!!!!!!!!!! ( test_expression: 16 )
!!!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!!!!!!!! ( classical_var_items: 46 )
!!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :ORD
!!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :DATE
!!!!!!!!!!!!!!!! ( eq: 136 )
!!!!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!!!!! ( unsigned_integer: -139 ) VALEUR :310885
!!!!!!!!!!!!!! ( statements: 5 )
!!!!!!!!!!!!!!! ( statement: 62 )
!!!!!!!!!!!!!!!! ( for_st: 11 )
!!!!!!!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :OL
!!!!!!!!!!!!!!!!!! ( collection_expression: 90 )
!!!!!!!!!!!!!!!!!!! ( DB_object_set: 39 )
!!!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :ORDER_LINE
!!!!!!!!!!!!!!!!!!!! ( coll_cond_primary: 95 )
!!!!!!!!!!!!!!!!!!!!! ( relation_condition: 41 )
!!!!!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :OR_OL
!!!!!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :ORD
!!!!!!!!!!!!!!!!!! ( statements: 5 )
!!!!!!!!!!!!!!!!!!! ( statement: 62 )
!!!!!!!!!!!!!!!!!!!! ( for_st: 11 )
!!!!!!!!!!!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :PROD
!!!!!!!!!!!!!!!!!!!!!! ( collection_expression: 90 )
!!!!!!!!!!!!!!!!!!!!!!! ( DB_object_set: 39 )
!!!!!!!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :PRODUCT
!!!!!!!!!!!!!!!!!!!!!!!! ( coll_cond_primary: 95 )
!!!!!!!!!!!!!!!!!!!!!!!!! ( relation_condition: 41 )
!!!!!!!!!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :OL_P
!!!!!!!!!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :OL
!!!!!!!!!!!!!!!!!!!!!! ( statements: 5 )
!!!!!!!!!!!!!!!!!!!!!!! ( statement: 62 )
```

```
!!!!!!!!!!!!!!!!!!!! ( if_st: 31 )
!!!!!!!!!!!!!!!!!!!!! ( general_primary: 81 )
!!!!!!!!!!!!!!!!!!!!! ( test_expression: 16 )
!!!!!!!!!!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!!!!!!!!!!!!! ( classical_var_items: 46 )
!!!!!!!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :PROD
!!!!!!!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :NBPRO
!!!!!!!!!!!!!!!!!!!!!! ( eq: 136 )
!!!!!!!!!!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!!!!!!!!! ( unsigned_integer: -139 ) VALEUR :5159
!!!!!!!!!!!!!!!!!!!!! ( statements: 5 )
!!!!!!!!!!!!!!!!!!!!!! ( statement: 62 )
!!!!!!!!!!!!!!!!!!!!!! ( ass_st: 29 )
!!!!!!!!!!!!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :CNBCUST
!!!!!!!!!!!!!!!!!!!!!! ( assign_expression: 76 )
!!!!!!!!!!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!!!!!!!!!!! ( add_arith: 17 )
!!!!!!!!!!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :CNBCUST
!!!!!!!!!!!!!!!!!!!!!!! ( plus: 122 )
!!!!!!!!!!!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!!!!!!!!!!! ( unsigned_integer: -139 ) VALEUR :1
!!!!!!!!!!!!!!!!!!!!! ( statement: 62 )
!!!!!!!!!!!!!!!!!!!!! ( next_name_st: 63 )
!!!!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :CUST
!!!!!!!!!!!!!!!!!!!!! ( statement: 62 )
!!!!!!!!!!!!!!!!!!!!! ( dummy: 116 )
```

```
      ************************************
      *                                  *
      *   COBOL TEXT GENERATED FOR       *
      *   THE EXAMPLE OF CHAPTER 8       *
      *                                  *
      ************************************

************************
IDENTIFICATION DIVISION.
************************

PROGRAM-ID.       FUNC1NBCUST.

**********************
ENVIRONMENT DIVISION.
**********************

CONFIGURATION SECTION.

SOURCE-COMPUTER. VAX-VMS.
OBJECT-COMPUTER. VAX-VMS.

***************
DATA DIVISION.
***************

WORKING-STORAGE SECTION.

*-------------------------------------------------------------
* LIST OF THE PARAMETERS FOR THE INTERFACE WITH THE GAM
*-------------------------------------------------------------
01   DBSTAT.
     02   FNCODE; PIC 99.
     02   ERRCODE; PIC 99.
01   Z-CODES.
     02   OPCODE; PIC 99; USAGE COMP.
     02   GETCODE; PIC 9.
     02   RECTCODE; PIC 99; USAGE COMP.
     02   RECREF; PIC S9(10); USAGE COMP.
     02   KEYCODE; PIC 99; USAGE COMP.
     02   OPERATOR; PIC 9.
01   Z-VALUES.
     02   Z-VALUE; PIC X(84).
     02   Z-VALUE-CUSTOMER; REDEFINES Z-VALUE.
          03   NB-CUS; PIC S9(10); USAGE COMP.
          03   NAME; PIC X(20).
          03   ADDRESS.
               04   NBER; PIC S9(4); USAGE COMP.
               04   STREET; PIC X(30).
               04   CITY; PIC X(20).
     02   Z-VALUE-ORDER; REDEFINES Z-VALUE.
          03   NB-ORD; PIC S9(10); USAGE COMP.
          03   DATE-O; PIC 9(6).
          03   FILLER; PIC X(68).
     02   Z-VALUE-ORDER-LINE; REDEFINES Z-VALUE.
          03   Q; PIC S9(10); USAGE COMP.
          03   FILLER; PIC X(74).
```

```
    02  Z-VALUE-PRODUCT; REDEFINES Z-VALUE.
        03  NB-PRO; PIC S9(10); USAGE COMP.
        03  PRICE; PIC S9(8)V9(8); USAGE COMP.
        03  Q-STK; PIC S9(10); USAGE COMP.
        03  FILLER; PIC X(44).
01  Z-PATHS.
    02  PATHLIST.
        03  PATHCODE; PIC 9(4); USAGE COMP; OCCURS  8 TIMES.
    02  CURLIST.
        03  CURORIGIN; PIC S9(10); USAGE COMP; OCCURS  8 TIMES.

*----------------------------------------
* CODES OF THE ACTIONS OFFERED BY THE GAM
*----------------------------------------
01  OPEN-DB; PIC 99; USAGE COMP; VALUE 01.
01  CLOSE-DB; PIC 99; USAGE COMP; VALUE 02.
01  SEQ-ACCESS; PIC 99; USAGE COMP; VALUE 03.
01  KEY-ACCESS; PIC 99; USAGE COMP; VALUE 04.
01  SEQ-AC-W-PATH; PIC 99; USAGE COMP; VALUE 05.
01  KEY-AC-W-PATH; PIC 99; USAGE COMP; VALUE 06.
01  REF-ACCESS; PIC 99; USAGE COMP; VALUE 07.
01  CREATE-REC; PIC 99; USAGE COMP; VALUE 08.
01  DELETE-REC; PIC 99; USAGE COMP; VALUE 09.
01  MODIFY-ITEMS; PIC 99; USAGE COMP; VALUE 10.
01  MODIFY-IKO; PIC 99; USAGE COMP; VALUE 11.
01  ATTACH-REC; PIC 99; USAGE COMP; VALUE 12.
01  DETACH-REC; PIC 99; USAGE COMP; VALUE 12.
01  TRANSFER-REC; PIC 99; USAGE COMP; VALUE 12.
01  REF-COMP; PIC 99; USAGE COMP; VALUE 20.
01  REF-NULL; PIC 99; USAGE COMP; VALUE 21.
01  REF-ASSIGN; PIC 99; USAGE COMP; VALUE 22.

*----------------------------
* CODES OF THE RECORD TYPES
*----------------------------
01  CUSTOMER-CODE; PIC 99; USAGE COMP; VALUE 14.
01  ORDER-CODE; PIC 99; USAGE COMP; VALUE 18.
01  ORDER-LINE-CODE; PIC 99; USAGE COMP; VALUE 15.
01  PRODUCT-CODE; PIC 99; USAGE COMP; VALUE 11.

*--------------------------------
* CODES OF THE ACCESS PATH TYPES
*--------------------------------
01  C-OR; PIC 99; USAGE COMP; VALUE 26.
01  OR-OL; PIC 99; USAGE COMP; VALUE 25.
01  OL-P; PIC 99; USAGE COMP; VALUE 23.

*------------------
* CODES OF THE KEYS
*------------------

*--------------------------------------
* CURRENT INSTANCE OF THE RECORD TYPES
*--------------------------------------
01  CUR-CUST ;PIC S9(10); USAGE COMP.
01  CUR-ORD ;PIC S9(10); USAGE COMP.
01  CUR-OL ;PIC S9(10); USAGE COMP.
01  CUR-PROD ;PIC S9(10); USAGE COMP.
```

```
*----------------
* OTHER VARIABLES
*----------------
01  CNBCUST; PIC S9(10); USAGE COMP.
01  DBALOPEN; PIC 9.
01  CUST.
    02  CUR; PIC S9(10); USAGE COMP.
    02  TYP; PIC X(10).
    02  FIL; PIC X(10).
    02  ITEMS.
        03  NB-CUS; PIC S9(10); USAGE COMP.
        03  NAME; PIC X(20).
        03  ADDRESS.
            04  NBER; PIC S9(4); USAGE COMP.
            04  STREET; PIC X(30).
            04  CITY; PIC X(20).
01  ORD.
    02  CUR; PIC S9(10); USAGE COMP.
    02  TYP; PIC X(10).
    02  FIL; PIC X(10).
    02  ITEMS.
        03  NB-ORD; PIC S9(10).
        03  DATE-O; PIC S9(6).
01  OL.
    02  CUR; PIC S9(10); USAGE COMP.
    02  TYP; PIC X(10).
    02  FIL; PIC X(10).
    02  ITEMS.
        03  Q; PIC S9(10); USAGE COMP.
01  PROD.
    02  CUR; PIC S9(10); USAGE COMP.
    02  TYP; PIC X(10).
    02  FIL; PIC X(10).
    02  ITEMS.
        03  NB-PRO; PIC S9(10); USAGE COMP.
        03  PRICE; PIC S9(8)V9(8); USAGE COMP.
        03  Q-STK; PIC S9(10); USAGE COMP.

**********************
PROCEDURE DIVISION.
**********************
MAIN-PROGRAM.
    PERFORM OVERTURE-DB.
    GO TO MAIN-TRT.
ENDST.
    PERFORM CLOTURE-DB.
    STOP RUN.
OVERTURE-DB.
    MOVE 0 TO DBALOPEN.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE OPEN-DB TO OPCODE.
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE = 2; MOVE 1 TO DBALOPEN;
      ELSE IF ERRCODE NOT = 0; GO TO TR-ERROR.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
```

```
    MOVE REF-NULL TO OPCODE.
    MOVE CUR-CUST TO CURORIGIN(1).
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE NOT = 0; GO TO TR-ERROR.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE REF-NULL TO OPCODE.
    MOVE CUR-ORD TO CURORIGIN(1).
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE NOT = 0; GO TO TR-ERROR.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE REF-NULL TO OPCODE.
    MOVE CUR-OL TO CURORIGIN(1).
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE NOT = 0; GO TO TR-ERROR.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE REF-NULL TO OPCODE.
    MOVE CUR-PROD TO CURORIGIN(1).
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE NOT = 0; GO TO TR-ERROR.
CLOTURE-DB.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE CLOSE-DB TO OPCODE.
    IF DBALOPEN = 0;
        CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS;
        IF ERRCODE NOT = 0; GO TO TR-ERROR.
MAIN-TRT.
    MOVE 0 TO CNBCUST.
    GO TO TR-FORDB-1.
END-FORDB-1.
    GO TO ENDST.

TR-FORDB-1.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE SEQ-ACCESS TO OPCODE.
    MOVE 1 TO GETCODE.
    MOVE CUSTOMER-CODE TO RECTCODE.
    MOVE CUR-CUST TO RECREF.
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES,Z-PATHS.
    IF ERRCODE NOT = 0;
       IF ERRCODE NOT = 1; GO TO TR-ERROR;
         ELSE GO TO END-TRFORDB-1.
    MOVE RECREF TO CUR-CUST.
    MOVE CORRESPONDING Z-VALUE-CUSTOMER TO CUST.
    GO TO TR-FORDBST-1.
END-TRFORDBST-1.
    GO TO TR-FORDB-1.
END-TRFORDB-1.
    GO TO END-FORDB-1.
TR-FORDBST-1.
    GO TO TR-FORDB-2.
END-FORDB-2.
    GO TO END-TRFORDBST-1.
```

```
TR-FORDB-2.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE SEQ-AC-W-PATH TO OPCODE.
    MOVE 1 TO GETCODE.
    MOVE ORDER-CODE TO RECTCODE.
    MOVE CUR-ORD TO RECREF.
    MOVE C-OR TO PATHCODE(1).
    MOVE CUR-CUST TO CURORIGIN(1).
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES,Z-PATHS.
    IF ERRCODE NOT = 0;
      IF ERRCODE NOT = 1; GO TO TR-ERROR;
        ELSE GO TO END-TRFORDB-2.
    MOVE RECREF TO CUR-ORD.
    MOVE CORRESPONDING Z-VALUE-ORDER TO ORD.
    GO TO TR-FORDBST-2.
END-TRFORDBST-2.
    GO TO TR-FORDB-2.
END-TRFORDB-2.
    GO TO END-FORDB-2.
TR-FORDBST-2.
    IF DATE-O OF ORD IS = 310885;
      GO TO TH-ST-1.
END-IF-1.
    GO TO END-TRFORDBST-2.
TH-ST-1.
    GO TO TR-FORDB-3.
END-FORDB-3.
    GO TO END-IF-1.
TR-FORDB-3.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE SEQ-AC-W-PATH TO OPCODE.
    MOVE 1 TO GETCODE.
    MOVE ORDER-LINE-CODE TO RECTCODE.
    MOVE CUR-OL TO RECREF.
    MOVE OR-OL TO PATHCODE(1).
    MOVE CUR-ORD TO CURORIGIN(1).
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES,Z-PATHS.
    IF ERRCODE NOT = 0;
      IF ERRCODE NOT = 1; GO TO TR-ERROR;
        ELSE GO TO END-TRFORDB-3.
    MOVE RECREF TO CUR-OL.
    MOVE CORRESPONDING Z-VALUE-ORDER-LINE TO OL.
    GO TO TR-FORDBST-3.
END-TRFORDBST-3.
    GO TO TR-FORDB-3.
END-TRFORDB-3.
    GO TO END-FORDB-3.
TR-FORDBST-3.
    GO TO TR-FORDB-4.
END-FORDB-4.
    GO TO END-TRFORDBST-3.
```

```
TR-FORDB-4.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE SEQ-AC-W-PATH TO OPCODE.
    MOVE 1 TO GETCODE.
    MOVE PRODUCT-CODE TO RECTCODE.
    MOVE CUR-PROD TO RECREF.
    MOVE OL-P TO PATHCODE(1).
    MOVE CUR-OL TO CURORIGIN(1).
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES,Z-PATHS.
    IF ERRCODE NOT = 0;
      IF ERRCODE NOT = 1; GO TO TR-ERROR;
         ELSE GO TO END-TRFORDB-4.
    MOVE RECREF TO CUR-PROD.
    MOVE CORRESPONDING Z-VALUE-PRODUCT TO PROD.
    GO TO TR-FORDBST-4.
END-TRFORDBST-4.
    GO TO TR-FORDB-4.
END-TRFORDB-4.
    GO TO END-FORDB-4.
TR-FORDBST-4.
    IF NB-PRO IN PROD IS = 5159;
      GO TO TH-ST-2.
END-IF-2.
    GO TO END-TRFORDBST-4.
TH-ST-2.
    COMPUTE CNBCUST = CNBCUST + 1.
    GO TO END-TRFORDBST-1.
    GO TO END-IF-2.
TR-ERROR.
    DISPLAY 'ERROR IN DATABASE'.
    DISPLAY DBSTAT.
    STOP RUN.
```

****
*
* DECOMPILED TEXT, HIERARCHICAL *
*   TREE AND COBOL PROGRAM FOR   *
*   THE FIRST NEW EXAMPLE   *
*
****

```
      ************************************
      *                                  *
      *  DECOMPILED TEXT FOR THE FIRST   *
      *     EXAMPLE OF THE APPENDIX      *
      *                                  *
      ************************************


 (*debut adl *)
 algorithm EX1
 type INT = integer

 var I, J : INT;
     MESS_ERR : string( 30);
     PRES : boolean

 begin I := 0;
       PRES := false;
       GETNB!( J);
       while( I <> 100) and( PRES = false) do
            I := I + 1;
            if( J = I ^ 2)
                then PRES := true
            endif;

      endwhile;
      if PRES = false
          then MESS_ERR := 'le nombre donne n est pas un carre d e
tier';
               MESS!( MESS_ERR)
      endif;

end.
```

```
      ************************************
      *                                  *
      *  HIERARCHICAL TREE FOR THE       *
      *  FIRST EXAMPLE OF THE APPENDIX   *
      *                                  *
      ************************************

!annot
!!( formalisme : -1 ) adl
!! ( point_d_entree: 9 )
!!! ( name: -138 ) VALEUR :EX1
!!! ( intern_declaration_part: 23 )
!!!! ( type_declar: 48 )
!!!!! ( type_declarations: 1 )
!!!!!! ( type_declaration: 24 )
!!!!!!! ( name: -138 ) VALEUR :INT
!!!!!!! ( simple: 52 )
!!!!!!!! ( integer: 114 )
!!!! ( var_declar: 50 )
!!!!! ( variable_declarations: 2 )
!!!!!! ( variable_declaration: 25 )
!!!!!!! ( names: 3 )
!!!!!!!! ( name: -138 ) VALEUR :I
!!!!!!!! ( name: -138 ) VALEUR :J
!!!!!!! ( name: -138 ) VALEUR :INT
!!!!!! ( variable_declaration: 25 )
!!!!!!! ( names: 3 )
!!!!!!!! ( name: -138 ) VALEUR :MESS_ERR
!!!!!!! ( simple: 52 )
!!!!!!!! ( str: 53 )
!!!!!!!!! ( unsigned_integer: -139 ) VALEUR :30
!!!!!! ( variable_declaration: 25 )
!!!!!!! ( names: 3 )
!!!!!!!! ( name: -138 ) VALEUR :PRES
!!!!!!! ( simple: 52 )
!!!!!!!! ( boolean: 112 )
!!! ( statement_part: 61 )
!!!! ( statements: 5 )
!!!!! ( statement: 62 )
!!!!!! ( ass_st: 29 )
!!!!!!! ( variable: 101 )
!!!!!!!! ( name: -138 ) VALEUR :I
!!!!!!! ( assign_expression: 76 )
!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!! ( primary: 86 )
!!!!!!!!!! ( unsigned_integer: -139 ) VALEUR :0
!!!!! ( statement: 62 )
!!!!!! ( ass_st: 29 )
!!!!!!! ( variable: 101 )
!!!!!!!! ( name: -138 ) VALEUR :PRES
!!!!!!! ( assign_expression: 76 )
!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!! ( primary: 86 )
!!!!!!!!!! ( false: 130 )
!!!!! ( statement: 62 )
!!!!!! ( call_st: 32 )
!!!!!!! ( proc_name: -140 ) VALEUR :GETNBI
!!!!!!! ( param: 67 )
```

```
!!!!!!!! ( param_list: 6 )
!!!!!!!!! ( variable: 101 )
!!!!!!!!!! ( name: -138 ) VALEUR :J
!!!!! ( statement: 62 )
!!!!!! ( while_st: 30 )
!!!!!!! ( and_expression: 36 )
!!!!!!!! ( general_primary: 81 )
!!!!!!!!! ( gen_expr_parenth: 82 )
!!!!!!!!!! ( general_primary: 81 )
!!!!!!!!!!! ( test_expression: 16 )
!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :I
!!!!!!!!!!!! ( not_eq: 135 )
!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!! ( unsigned_integer: -139 ) VALEUR :100
!!!!!!!! ( general_primary: 81 )
!!!!!!!!! ( gen_expr_parenth: 82 )
!!!!!!!!!! ( general_primary: 81 )
!!!!!!!!!!! ( test_expression: 16 )
!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :PRES
!!!!!!!!!!!! ( eq: 136 )
!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!! ( false: 130 )
!!!!!!! ( statements: 5 )
!!!!!!!! ( statement: 62 )
!!!!!!!!! ( ass_st: 29 )
!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!! ( name: -138 ) VALEUR :I
!!!!!!!!!! ( assign_expression: 76 )
!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!! ( add_arith: 17 )
!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :I
!!!!!!!!!!!!! ( plus: 122 )
!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!! ( unsigned_integer: -139 ) VALEUR :1
!!!!!!!! ( statement: 62 )
!!!!!!!!! ( if_st: 31 )
!!!!!!!!!! ( general_primary: 81 )
!!!!!!!!!!! ( gen_expr_parenth: 82 )
!!!!!!!!!!!! ( general_primary: 81 )
!!!!!!!!!!!!! ( test_expression: 16 )
!!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :J
!!!!!!!!!!!!!!! ( eq: 136 )
!!!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!!!! ( exp_term: 19 )
```

```
!!!!!!!!!!!!!!!! ( primary: 36 )
!!!!!!!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :I
!!!!!!!!!!!!!!!!! ( exp: 120 )
!!!!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!!!!! ( unsigned_integer: -139 ) VALEUR :2
!!!!!!!!!! ( statements: 5 )
!!!!!!!!!! ( statement: 62 )
!!!!!!!!!!! ( ass_st: 29 )
!!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!!! ( name: -138 ) VALEUR :PRES
!!!!!!!!!!!!! ( assign_expression: 76 )
!!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!!! ( true: 129 )
!!!!!!!! ( statement: 62 )
!!!!!!!!! ( dummy: 116 )
!!!!! ( statement: 62 )
!!!!!! ( if_st: 31 )
!!!!!!! ( general_primary: 31 )
!!!!!!!! ( test_expression: 16 )
!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!! ( name: -138 ) VALEUR :PRES
!!!!!!!!! ( eq: 136 )
!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!! ( false: 130 )
!!!!!!! ( statements: 5 )
!!!!!!!! ( statement: 62 )
!!!!!!!!! ( ass_st: 29 )
!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!! ( name: -138 ) VALEUR :MESS_ERR
!!!!!!!!!! ( assign_expression: 76 )
!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!! ( string: -137 ) VALEUR :'le nombre donne n est pas
n carre d entier'
!!!!!!!! ( statement: 62 )
!!!!!!!!! ( call_st: 32 )
!!!!!!!!!! ( proc_name: -140 ) VALEUR :MESS!
!!!!!!!!!! ( param: 67 )
!!!!!!!!!!! ( param_list: 6 )
!!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!!! ( name: -138 ) VALEUR :MESS_ERR
!!!!! ( statement: 62 )
!!!!!! ( dummy: 116 )
```

```
      ********************************
      *                              *
      *  COBOL TEXT FOR THE FIRST    *
      *  EXAMPLE OF THE APPENDIX     *
      *                              *
      ********************************


************************
IDENTIFICATION DIVISION.
************************

PROGRAM-ID.       EX1.

**********************
ENVIRONMENT DIVISION.
**********************

CONFIGURATION SECTION.

SOURCE-COMPUTER. VAX-VMS.
OBJECT-COMPUTER. VAX-VMS.

***************
DATA DIVISION.
***************

WORKING-STORAGE SECTION.

*-------------------------------------------------------------
* LIST OF THE PARAMETERS FOR THE INTERFACE WITH THE GAM
*-------------------------------------------------------------
01   DBSTAT.
     02   FNCODE; PIC 99.
     02   ERRCODE; PIC 99.
01   Z-CODES.
     02   OPCODE; PIC 99; USAGE COMP.
     02   SETCODE; PIC 9.
     02   RECTCODE; PIC 99; USAGE COMP.
     02   RECREF; PIC S9(10); USAGE COMP.
     02   KEYCODE; PIC 99; USAGE COMP.
     02   OPERATOR; PIC 9.
01   Z-VALUES.
     02   Z-VALUE; PIC X(1).
01   Z-PATHS.
     02   PATHLIST.
          03   PATHCODE; PIC 9(4); USAGE COMP; OCCURS  8 TIMES.
     02   CURLIST.
          03   CURORIGIN; PIC S9(10); USAGE COMP; OCCURS  8 TIMES.

*-----------------------------------------
* CODES OF THE ACTIONS OFFERED BY THE GAM
*-----------------------------------------
01   OPEN-DB; PIC 99; USAGE COMP; VALUE 01.
01   CLOSE-DB; PIC 99; USAGE COMP; VALUE 02.
01   SEQ-ACCESS; PIC 99; USAGE COMP; VALUE 03.
01   KEY-ACCESS; PIC 99; USAGE COMP; VALUE 04.
01   SEQ-AC-W-PATH; PIC 99; USAGE COMP; VALUE 05.
01   KEY-AC-W-PATH; PIC 99; USAGE COMP; VALUE 06.
```

```
01  REF-ACCESS; PIC 99; USAGE COMP; VALUE 07.
01  CREATE-REC; PIC 99; USAGE COMP; VALUE 08.
01  DELETE-REC; PIC 99; USAGE COMP; VALUE 09.
01  MODIFY-ITEMS; PIC 99; USAGE COMP; VALUE 10.
01  MODIFY-IKC; PIC 99; USAGE COMP; VALUE 11.
01  ATTACH-REC; PIC 99; USAGE COMP; VALUE 12.
01  DETACH-REC; PIC 99; USAGE COMP; VALUE 12.
01  TRANSFER-REC; PIC 99; USAGE COMP; VALUE 12.
01  REF-COMP; PIC 99; USAGE COMP; VALUE 20.
01  REF-NULL; PIC 99; USAGE COMP; VALUE 21.
01  REF-ASSIGN; PIC 99; USAGE COMP; VALUE 22.

*--------------------------
* CODES OF THE RECORD TYPES
*--------------------------


*-------------------------------
* CODES OF THE ACCESS PATH TYPES
*-------------------------------


*------------------
* CODES OF THE KEYS
*------------------


*---------------------------------------
* CURRENT INSTANCE OF THE RECORD TYPES
*---------------------------------------


*----------------
* OTHER VARIABLES
*----------------
01  I; PICTURE IS S9(10); USAGE COMP.
01  J; PICTURE IS S9(10); USAGE COMP.
01  MESS-ERR; PICTURE IS X(30).
01  PRES; PICTURE IS A.
01  DBALOPEN; PICTURE IS 9.

********************
PROCEDURE DIVISION.
********************
MAIN-PROGRAM.
    PERFORM OVERTURE-DB.
    GO TO MAIN-TRT.
ENDST.
    PERFORM CLOTURE-DB.
    STOP RUN.
OVERTURE-DB.
    MOVE 0 TO DBALOPEN.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE OPEN-DB TO OPCODE.
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE = 2; MOVE 1 TO DBALOPEN;
      ELSE IF ERRCODE NOT = 0; GO TO TR-ERROR.
```

```
CLOTURE-DB.
     MOVE ZEROS TO Z-CODES.
     MOVE SPACES TO Z-VALUES, Z-PATHS.
     MOVE CLOSE-DB TO OPCODE.
     IF DBALOPEN = 0;
          CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS;
          IF ERRCODE NOT = 0; GO TO TR-ERROR.
MAIN-TRT.
     MOVE 0 TO I.
     MOVE "F" TO PRES.
     CALL "GETNB" USING J.
     GO TO WH-1.
WH-1.
     IF  (I IS NOT = 100)  AND  (PRES IS =  "F" ) ;
     GO TO WHST-1.
     IF PRES IS =  "F" ;
       GO TO TH-ST-1.
END-IF-1.
     GO TO ENDST.

WHST-1.
     COMPUTE I = I + 1.
     IF  (J IS = I ** 2) ;
       GO TO TH-ST-2.
END-IF-2.
     GO TO WH-1.
TH-ST-1.
     MOVE "le nombre donne n est pas un carre d entier" TO MESS-ERR.
     CALL "MESS" USING MESS-ERR.
     GO TO END-IF-1.
TH-ST-2.
     MOVE "T" TO PRES.
     GO TO END-IF-2.
TR-ERROR.
     DISPLAY 'ERROR IN DATABASE'.
     DISPLAY DBSTAT.
     STOP RUN.
```

****
*
* DECOMPILED TEXT, HIERARCHICAL
* TREE AND COBOL PROGRAM FOR
* THE SECOND NEW EXAMPLE
*
****

```
                ******************************
                *                            *
                *  DECOMPILED TEXT FOR THE   *
                *     SECOND NEW EXAMPLE     *
                *                            *
                ******************************

(*debut adl *)
algorithm EX2

var CUST : ref of CUSTOMER;
    NBER_TP : numeric( 4, 0)

begin for CUST := CUSTOMER () do
          if( ( CUST). CITY = 'NAMUR') and( ( CUST). STREET = 'rue
e fer')
               then NBER_TP := ( CUST). NBER;
                    modify CUST(: NBER= NBER_TP + 2)
               else if( ( CUST). CITY = 'BRUSSELS') and( ( CUST). S
REET = 'rue du sablon')
                         then delete CUST
                    endif
          endif
      endfor
end.
```

```
                ************************************
                *                                  *
                *    HIERARCHICAL TREE FOR THE     *
                *  SECOND EXAMPLE OF THE APPENDIX  *
                *                                  *
                ************************************

! annot
!! ( formalisme : -1 ) adl
!! ( point_d_entree: 9 )
!!! ( name: -138 ) VALEUR :EX2
!!! ( intern_declaration_part: 23 )
!!!! ( notype: 110 )
!!!! ( var_declar: 50 )
!!!!! ( variable_declarations: 2 )
!!!!!! ( variable_declaration: 25 )
!!!!!!! ( names: 3 )
!!!!!!!! ( name: -138 ) VALEUR :CUST
!!!!!!! ( structured: 54 )
!!!!!!!! ( ref_name: 60 )
!!!!!!!!! ( name: -138 ) VALEUR :CUSTOMER
!!!!!! ( variable_declaration: 25 )
!!!!!!! ( names: 3 )
!!!!!!!! ( name: -138 ) VALEUR :NBER_TP
!!!!!!! ( simple: 52 )
!!!!!!!! ( numeric: 26 )
!!!!!!!!! ( unsigned_integer: -139 ) VALEUR :4
!!!!!!!!! ( unsigned_integer: -139 ) VALEUR :0
!!! ( statement_part: 61 )
!!!! ( statements: 5 )
!!!!! ( statement: 62 )
!!!!!! ( for_st: 11 )
!!!!!!! ( variable: 101 )
!!!!!!!! ( name: -138 ) VALEUR :CUST
!!!!!!! ( collection_expression: 90 )
!!!!!!!! ( DB_object_set: 89 )
!!!!!!!!! ( name: -138 ) VALEUR :CUSTOMER
!!!!!!!!! ( coll_cond_primary: 95 )
!!!!!!!!!! ( nopredicate: 126 )
!!!!!!! ( statements: 5 )
!!!!!!!! ( statement: 62 )
!!!!!!!!! ( if_else_st: 12 )
!!!!!!!!!! ( and_expression: 36 )
!!!!!!!!!!! ( general_primary: 81 )
!!!!!!!!!!!! ( gen_expr_parenth: 82 )
!!!!!!!!!!!!! ( general_primary: 81 )
!!!!!!!!!!!!!! ( test_expression: 16 )
!!!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!!!!!!!! ( classical_var_items: 46 )
!!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :CUST
!!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :CITY
!!!!!!!!!!!!!!!! ( eq: 136 )
!!!!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!!!! ( string: -137 ) VALEUR :'NAMUR'
```

```
!!!!!!!!!! ( general_primary: 81 )
!!!!!!!!!!! ( gen_expr_parenth: 82 )
!!!!!!!!!!!! ( general_primary: 81 )
!!!!!!!!!!!!! ( test_expression: 16 )
!!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!!!!!!! ( classical_var_items: 46 )
!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :CUST
!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :STREET
!!!!!!!!!!!!!!!! ( eq: 136 )
!!!!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!!!!! ( string: -137 ) VALEUR :'rue de fer'
!!!!!!!!!! ( statements: 5 )
!!!!!!!!!!! ( statement: 62 )
!!!!!!!!!!!! ( ass_st: 29 )
!!!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!!!! ( name: -138 ) VALEUR :NBER_TP
!!!!!!!!!!!!! ( assign_expression: 76 )
!!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!!!!!!! ( classical_var_items: 46 )
!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :CUST
!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :NBER
!!!!!!!!!!! ( statement: 62 )
!!!!!!!!!!!! ( db_mod: 68 )
!!!!!!!!!!!!! ( modif: 33 )
!!!!!!!!!!!!!! ( name: -138 ) VALEUR :CUST
!!!!!!!!!!!!!! ( modif_conds: 72 )
!!!!!!!!!!!!!!! ( item_cond: 14 )
!!!!!!!!!!!!!!!! ( no_name: 127 )
!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :NBER
!!!!!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!!!!!! ( add_arith: 17 )
!!!!!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :NBER_TP
!!!!!!!!!!!!!!!!!! ( plus: 122 )
!!!!!!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!!!!!! ( unsigned_integer: -139 ) VALEUR :2
!!!!!!!!!! ( statements: 5 )
!!!!!!!!!!! ( statement: 62 )
!!!!!!!!!!!! ( if_st: 31 )
!!!!!!!!!!!!! ( and_expression: 36 )
!!!!!!!!!!!!!! ( general_primary: 81 )
!!!!!!!!!!!!!!! ( gen_expr_parenth: 82 )
!!!!!!!!!!!!!!!! ( general_primary: 81 )
!!!!!!!!!!!!!!!!! ( test_expression: 16 )
!!!!!!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!!!!!!!!!!! ( classical_var_items: 46 )
!!!!!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :CUST
!!!!!!!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :CITY
```

```
!!!!!!!!!!!!!!!!!! ( eq: 135 )
!!!!!!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!!!!!! ( string: -137 ) VALEUR :'BRUSSELS'
!!!!!!!!!!!!!! ( general_primary: 81 )
!!!!!!!!!!!!!!! ( gen_expr_parenth: 82 )
!!!!!!!!!!!!!!!! ( general_primary: 81 )
!!!!!!!!!!!!!!!!! ( test_expression: 16 )
!!!!!!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!!!!!!! ( variable: 101 )
!!!!!!!!!!!!!!!!!!!!! ( classical_var_items: 46 )
!!!!!!!!!!!!!!!!!!!!!! ( name: -133 ) VALEUR :CUST
!!!!!!!!!!!!!!!!!!!!!! ( name: -133 ) VALEUR :STREET
!!!!!!!!!!!!!!!!!! ( eq: 136 )
!!!!!!!!!!!!!!!!!! ( arithmetic_expression: 83 )
!!!!!!!!!!!!!!!!!!! ( primary: 86 )
!!!!!!!!!!!!!!!!!!! ( string: -137 ) VALEUR :'rue du sablon'
!!!!!!!!!!!!! ( statements: 5 )
!!!!!!!!!!!!!! ( statement: 62 )
!!!!!!!!!!!!!!! ( db_mod: 68 )
!!!!!!!!!!!!!!!! ( del: 75 )
!!!!!!!!!!!!!!!! ( name: -138 ) VALEUR :CUST
```

```
      ************************************
      *                                  *
      *   COBOL TEXT FOR THE SECOND      *
      *     EXAMPLE OF THE APPENDIX      *
      *                                  *
      ************************************

************************
IDENTIFICATION DIVISION.
************************

PROGRAM-ID.       EX2.

**********************
ENVIRONMENT DIVISION.
**********************

CONFIGURATION SECTION.

SOURCE-COMPUTER. VAX-VMS.
OBJECT-COMPUTER. VAX-VMS.

***************
DATA DIVISION.
***************

WORKING-STORAGE SECTION.

*-------------------------------------------------------
* LIST OF THE PARAMETERS FOR THE INTERFACE WITH THE GAM
*-------------------------------------------------------
01   DBSTAT.
     02   FNCODE; PIC 99.
     02   ERRCODE; PIC 99.
01   Z-CODES.
     02   OPCODE; PIC 99; USAGE COMP.
     02   GETCODE; PIC 9.
     02   RECTCODE; PIC 99; USAGE COMP.
     02   RECREF; PIC S9(10); USAGE COMP.
     02   KEYCODE; PIC 99; USAGE COMP.
     02   OPERATOR; PIC 9.
01   Z-VALUES.
     02   Z-VALUE; PIC X(84).
     02   Z-VALUE-CUSTOMER; REDEFINES Z-VALUE.
          03   NB-CUS; PIC S9(10); USAGE COMP.
          03   NAME; PIC X(20).
          03   ADDRESS.
               04   NBER; PIC S9(4); USAGE COMP.
               04   STREET; PIC X(30).
               04   CITY; PIC X(20).
01   Z-PATHS.
     02   PATHLIST.
          03   PATHCODE; PIC 9(4); USAGE COMP; OCCURS  8 TIMES.
     02   CURLIST.
          03   CURORIGIN; PIC S9(10); USAGE COMP; OCCURS  8 TIMES.
```

```
*----------------------------------------
* CODES OF THE ACTIONS OFFERED BY THE GAM
*----------------------------------------
01  OPEN-DB; PIC 99; USAGE COMP; VALUE 01.
01  CLOSE-DB; PIC 99; USAGE COMP; VALUE 02.
01  SEQ-ACCESS; PIC 99; USAGE COMP; VALUE 03.
01  KEY-ACCESS; PIC 99; USAGE COMP; VALUE 04.
01  SEQ-AC-W-PATH; PIC 99; USAGE COMP; VALUE 05.
01  KEY-AC-W-PATH; PIC 99; USAGE COMP; VALUE 06.
01  REF-ACCESS; PIC 99; USAGE COMP; VALUE 07.
01  CREATE-REC; PIC 99; USAGE COMP; VALUE 08.
01  DELETE-REC; PIC 99; USAGE COMP; VALUE 09.
01  MODIFY-ITEMS; PIC 99; USAGE COMP; VALUE 10.
01  MODIFY-IKO; PIC 99; USAGE COMP; VALUE 11.
01  ATTACH-REC; PIC 99; USAGE COMP; VALUE 12.
01  DETACH-REC; PIC 99; USAGE COMP; VALUE 12.
01  TRANSFER-REC; PIC 99; USAGE COMP; VALUE 12.
01  REF-COMP; PIC 99; USAGE COMP; VALUE 20.
01  REF-NULL; PIC 99; USAGE COMP; VALUE 21.
01  REF-ASSIGN; PIC 99; USAGE COMP; VALUE 22.

*---------------------------
* CODES OF THE RECORD TYPES
*---------------------------
01  CUSTOMER-CODE; PIC 99; USAGE COMP; VALUE 14.

*--------------------------------
* CODES OF THE ACCESS PATH TYPES
*--------------------------------

*------------------
* CODES OF THE KEYS
*------------------

*--------------------------------------
* CURRENT INSTANCE OF THE RECORD TYPES
*--------------------------------------
01  CUR-CUST; PIC S9(10); USAGE COMP.

*----------------
* OTHER VARIABLES
*----------------
01  CUST.
    02  CUR; PIC S9(10); USAGE COMP.
    02  TYP; PIC X(10).
    02  FIL; PIC X(10).
    02  ITEMS.
        03  NB-CUS; PIC S9(10); USAGE COMP.
        03  NAME; PIC X(20).
        03  ADDRESS.
            04  NBER; PIC S9(4); USAGE COMP.
            04  STREET; PIC X(30).
            04  CITY; PIC X(20).
01  NBER-TP; PICTURE IS S9(4); USAGE COMP.
01  DBALOPEN; PICTURE IS 9.
```

```
*********************
PROCEDURE DIVISION.
*********************
MAIN-PROGRAM.
    PERFORM OVERTURE-DB.
    GO TO MAIN-TRT.
ENDST.
    PERFORM CLOTURE-DB.
    STOP RUN.
OVERTURE-DB.
    MOVE 0 TO DBALOPEN.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE OPEN-DB TO OPCODE.
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE = 2; MOVE 1 TO DBALOPEN;
      ELSE IF ERRCODE NOT = 0; GO TO TR-ERROR.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE REF-NULL TO OPCODE.
    MOVE CUR-CUST TO CURORIGIN(1).
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE NOT = 0; GO TO TR-ERROR.
CLOTURE-DB.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE CLOSE-DB TO OPCODE.
    IF DBALOPEN = 0;
        CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS;
        IF ERRCODE NOT = 0; GO TO TR-ERROR.
MAIN-TRT.
    GO TO TR-FORDB-1.
END-FORDB-1.
    GO TO ENDST.

TR-FORDB-1.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE SEQ-ACCESS TO OPCODE.
    MOVE 1 TO GETCODE.
    MOVE CUSTOMER-CODE TO RECTCODE.
    MOVE CUR-CUST TO RECREF.
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES,Z-PATHS.
    IF ERRCODE NOT = 0;
       IF ERRCODE NOT = 1; GO TO TR-ERROR;
         ELSE GO TO END-TRFORDB-1.
    MOVE RECREF TO CUR-CUST.
    MOVE CORRESPONDING Z-VALUE-CUSTOMER TO CUST.
    GO TO TR-FORDBST-1.
END-TRFORDBST-1.
    GO TO TR-FORDB-1.
END-TRFORDB-1.
    GO TO END-FORDB-1.
TR-FORDBST-1.
    IF  (CITY OF CUST IS = "NAMUR")  AND  (STREET OF CUST IS = "ru
 de fer") ;
            GO TO TH-ST-1;
     ELSE  GO TO EL-ST-1.
END-IF-1.
    GO TO END-TRFORDBST-1.
```

```
TH-ST-1.
    MOVE NBER OF CUST TO NBER-TP.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE MODIFY-ITEMS TO OPCODE.
    MOVE CUSTOMER-CODE TO RECTCODE.
    MOVE CUR-CUST TO RECREF.
    COMPUTE NBER OF Z-VALUE-CUSTOMER = NBER-TP + 2.
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE NOT = 0; GO TO TR-ERROR.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE MODIFY-IKD TO OPCODE.
    MOVE CUSTOMER-CODE TO RECTCODE.
    MOVE CUR-CUST TO RECREF.
    COMPUTE NBER OF Z-VALUE-CUSTOMER = NBER-TP + 2.
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE NOT = 0; GO TO TR-ERROR.
    GO TO END-IF-1.
EL-ST-1.
    IF  (CITY OF CUST IS = "BRUSSELS")  AND  (STREET OF CUST IS ="
rue du sablon") ;
      GO TO TH-ST-2.
END-IF-2.
    GO TO END-IF-1.
TH-ST-2.
    MOVE ZEROS TO Z-CODES.
    MOVE SPACES TO Z-VALUES, Z-PATHS.
    MOVE DELETE-REC TO OPCODE.
    MOVE CUSTOMER-CODE TO RECTCODE.
    MOVE CUR-CUST TO RECREF.
    CALL "DB" USING DBSTAT, Z-CODES, Z-VALUES, Z-PATHS.
    IF ERRCODE NOT = 0; GO TO TR-ERROR.
    MOVE RECREF TO CUR-CUST.
    GO TO END-IF-2.
TR-ERROR.
    DISPLAY 'ERROR IN DATABASE'.
    DISPLAY DBSTAT.
    STOP RUN.
```